PRESERVING PRIVACY WITH USER-CONTROLLED SHARING OF VERIFIED INFORMATION

A Thesis
Presented to
The Academic Faculty

by

David Allen Bauer

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2009

PRESERVING PRIVACY WITH USER-CONTROLLED SHARING OF VERIFIED INFORMATION

Approved by:

Professor Douglas M. Blough,
Committee Chair
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Professor Douglas M. Blough, Advisor
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Professor George F. Riley
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Professor Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Date Approved: 11 November 2009

To all the family, friends, teachers, and more who have shaped my life thus far

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Personal information, especially certified personal information, can be very valuable to its subject, but it can also be abused by other parties for identify theft, blackmail, fraud, and more. One partial solution to the problem is credentials, whereby personal information is tied to identity, for example by a photo or signature on a physical credential.

We present an efficient scheme for large, redactable, digital credentials that allow certified personal attributes to safely be used to provide identification. A novel method is provided for combining credentials, even when they were originally issued by different authorities. Compared to other redactable digital credential schemes, the proposed scheme is approximately two orders of magnitude faster, due to aiming for auditability over anonymity. In order to expand this scheme to hold other records, medical records for example, we present a method for efficient signatures on redactable data where there are dependencies between different pieces of data. Positive results are shown using both artificial datasets and a dataset derived from a Linux package manager.

Electronic credentials must of course be held in a physical device with electronic memory. To hedge against the loss or compromise of the physical device holding a user's credentials, the credentials may be split up. An architecture is developed and prototyped for using split-up credentials, with part of the credentials held by a network attached agent. This architecture is generalized into a framework for running identity agents with various capabilities. Finally, a system for securely sharing medical records is built upon the generalized agent framework. The medical records are optionally stored using the redactable digital credentials, for source verifiability.

CHAPTER I

INTRODUCTION

Personal information is increasingly used to establish identity and authorize transactions in the digital world. At the same time, identity theft and fraud, based on unauthorized disclosure and misuse of personal information, are rampant [27], and individuals are increasingly concerned about providing this information to every digital entity with which they establish a relationship.

The research described in this paper works to protect personal information, and is based on several key principles of identity management. First, people should have the maximum control possible over what personal information of theirs is disclosed in any given interaction. Second, more reliance can be placed on personal information that is verified by trusted third parties than on self-reported information. And third, if verified personal information is to be used, mechanisms to prevent that information from being copied and misused by unauthorized parties are essential.

Medical records are an example of data that is very personal, but also very valuable to the person that they are about, because they can save their subject's life.[1] However, such records do no good if they are not available to the subject and medical personnel treating the subject. Availability of medical records in an emergency is a particularly hard problem, because we don't know ahead of time who needs to access the records, when or where they will be accessed, nor what part of the records will be most useful. Additionally, the subject of the records will often be unconscious or otherwise incapable of helping.

An individual's medical records can also be valuable to others, to use against that

[1] Additionally, medical records are valuable in aggregate [54], but to aggregate medical or other records while preserving individual privacy is a different problem, not addressed in this work.

individual. For example, medical records might be used by an employer to (generally illegally) deny employment to somebody, based on conditions that might cause them to miss work or require expensive on-going care. Medical records could be used to embarrass a person, especially in the political arena, through a variety of possible issues (past drug or alcohol treatment, STD diagnosis, mental health reports, etc).

The research described in this paper includes work toward making medical records available when needed, while preventing their misuse.

1.0.1 Verified information

Information is an abstract thing. Information may be defined as, "knowledge communicated or received concerning a particular fact or circumstance" [48], leading to the question of what is knowledge? Knowledge may be defined as, "acquaintance with facts, truths, or principles" [49]. These definitions emphasis the idea of information being accurate. But information is often inaccurate, because of mistakes or purposeful deception.

Verified information is information for which there is evidence of its accuracy. Generally that evidence is simply an assertion by some entity that the information is true. If the entity is sufficiently trusted, such as assertion is enough to trust the information. Given the ease with which information can be fabricated, unverified information is irrelevant in this work. Unverified or fabricated information can be produced anywhere and at anytime, and so can't be controlled by the subject of it.

Source-verified information is information that is verified by the original source of the information. For many types of long-lived records, having the original creator of the record verify it through a digital signature removes the complexity of having trusted third parties investigate and assert the accuracy of the record later.

1.0.2 Minimal Information Disclosure

Privacy may be defined as "the ability of an individual or group to seclude themselves or information about themselves and thereby reveal themselves selectively". [66] Selectively releasing information is a fundamental basis for the research presented here. In particular, we want to release the minimal amount of information necessary to for the task at hand, and we want to disclose that information to the minimum set of entities required.

Minimal information disclosure in any given interaction is desirable from a user's perspective, and may even be necessary for a given technology to be widely adopted [14]. Clearly, if a user wants to release the minimum amount of personal information on a given interaction, this rules out a single credential approach, in which each user maintains one credential containing all of their personal information and uses that credential for every interaction. As in [14], we say that a user makes a claim about herself when she gives information about one or more personal attributes to a digital entity. Due to the wide variety of personal information that is used in digital interactions, the number of different possible claims is extremely large.

1.1 Problem Statement

The problem to be solved is, therefore, to provide an efficient and reliable mechanism that allows users to assert arbitrary subsets of verifiable claims over a sequence of interactions with different digital entities.

The focus of the work is on two situations.

1. The release of personal information in the process of identifying and authenticating oneself.

2. The sharing of medical records, especially with medical personnel in emergency cases.

1.2 Identity Management

Identity management, concerned with identifying and tracking entities, covers technologies and techniques for using and protecting personal information, especially verified information.

Identity management is a broad research area that is attracting a lot of attention currently. Major corporations working on this area include Microsoft with their "Vision for an Identity Metasystem" [40] which has lead to Microsoft Cardspace [16], Oracle with their conveniently named "Oracle Identity Management" product [46], Novell [43], Sun Microsystems [59], and many more.

A large range of products and services falls under the umbrella term of identity management. The part of identity management addressed in this work concerns the authorization of users, especially users who are not immediately connected with the system being accessed. There are also systems and products concerned with the detailed tracking of users more connected with the system being accessed; for example, tracking the actions of employees in a business or records of patients in a healthcare system. These systems are outside the scope of this work.

1.2.1 Terminology

The terminology used in this proposal will follow current standards in identity management for an asymmetric case. The entity being identified is the "user", while the entity the user's identity is being revealed to or proven to is the "relying party". An entity that is making an assertion about the identity of the user is an "identity provider". In the peer-to-peer case, each peer will be both a user and a relying party. (A peer can also act as an identity provider, but except for the special case when a user is providing self-certified information, this does not matter.) User attributes that the user is asserting are referred to as "claims".

1.2.2 Federated Identity Management

One widely used approach to identity management is federated identity management. In federated identity management, each user is part of a specific group (a federation). Identification and authentication within the federation is done on an individual/user basis. However, when a user wants to authenticate to a relying part in another federation, the user authenticates to their own federation (based on individual credentials), and then uses federation credentials to authenticate to the relying party's federation. The relying party's federation validates the user's credentials for the relying party. Federated identity management generally requires pre-arranged agreements between federations, and not a reliance on further third party identity providers. This helps to distinguish federated identity management from other forms of trusted-authority based authentication. A federation is generally a single corporation (or university), although finer-grain federations (e.g., departments) and hierarchies can also be done.

1.2.3 Credentials and claims

A credential is information accompanied by evidence for the validity of that information. Credentials are common in everyday life. We carry driver's licenses, employee and student IDs, credit cards, insurance cards, and more. We use user names or numbers along with passwords to login to e-mail, forums, and even ATMs. Physical keys are credentials used to open physical locks. Even our cars have credentials, in license plates, vehicle identification numbers, registration paperwork, and proof of insurance. All of these credentials assert information, although it may be very implicit, for example in the case of a physical key asserting that its bearer has the authority to open a certain lock.

Electronic credentials can be simple, like a user-name and password, or more complex, like a public key infrastructure (PKI) certificate. A PKI certificate is an electronic document that holds an identity and a public key, and that is signed by

a certificate authority (called the issuer). The owner holds the associated private key to prove that they are the legitimate holder of the certificate. The user-name and password combination is the most widely used scheme, because of its simplicity. However, this scheme also provides no direct information about the user. A user-name must be attached to a previously made account, and some other form of credential must be used to tie an identity to the account. Such accounts are very seldom shared between different domains, leading users to accumulate many different accounts, often with different user-names and passwords. PKI certificates for users are less common, but can solve the problem of needing to keep track of many different user-names and passwords.

Claims are statements, attributes, and/or data that are being asserted by an entity. A credential is structure with one or more claims about the entity presenting them combined with evidence for those claims. Claims can be complex in their content. For example, an address can include the country, state, county, city, zip code, and street name, without even getting to the level of a specific house or apartment. Claims may also have implicit, smaller claims that can be obtained from them. For example, it is common to show a full date of birth, just to extract the small claim that one is at least 21 years of age. We call small claims or pieces of a full claim "micro-claims". Many different micro-claims can be further derived from the single claim of age, e.g. the user is "at least 18", "at least 21", "at least 35", "at least 65", etc. In a more technical example, a set of micro-claims could consist of a large number of one-time use tokens, such as one-time use credit card numbers. The one-time use credit card numbers are pieces of the claim of owning the account to which they are connected.

We assume an architecture in which there are high-level, globally trusted identity providers that verify users' personal information and supply credentials that the users can give to service providers [40]. Considering the proliferation of trusted certificate

providers existing today, that assumption seems reasonable. Additionally, credential signatures can be hierarchical, whereby the identity provider that issues the credential passes along certification of their own trustworthiness certified by a higher certificate authority, e.g. in X.509. [29]

1.2.4 Offline and Online Credentials

Credentials can be described as online or offline credentials. [2] When a user shows a traditional credential, such as a driver's license, that showing is not recorded. There is no record at the DMV of when a driver's license was shown to prove the holder's name, age, or other details. By contrast, when a credit card is used, it is usually verified as valid (both in general, and in terms of having sufficient funds) with the issuer before the transaction is completed. The credit card issuer keeps a record of all of the times that the credit card is used (or even just checked). In this example, the driver's license is an off-line credential and the credit card is an on-line credential.

Credentials are often used for purposes outside of their original or stated intention. The most obvious example is a driver's license, which has became the de facto identity card in the US. Driver's licenses are so widely used, that legislation was passed in 2005 to require states to follow federal standards in issuing driver's licenses, including requiring that electronic records be available to all states [35]. [3] Credit cards are commonly used as a proof of adulthood, despite the proliferation of credit cards among teens. [33] Utility bills are often used as backup identification or as identification of one's current residence. [21]

1.3 Agents

Software agents are widely researched and used, but not clearly defined. For example, Franklin and Graesser provide nearly a dozen definitions in [24]. The Intelligent

[2] Technically, the *verification* of credentials is either online or offline; a credential can be both.
[3] Many states have passed legislation protesting the act and requiring noncompliance. [1]

Agents Project at IBM provides the general definition that will be used in this paper, "software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires." [5]

A mobile agent is a software agent that has the ability, "to transport itself from one machine to another" [24]. Alternately, instead of sending a whole agent to run on a remote system, an agent may send a piece of code to another system to be executed there. This model of remote code execution is actually widely used in the World Wide Web, in the form of Java applets, Javascript (ECMAScript, including AJAX), Macromedia Flash applications, and ActiveX controls. Note, however, that all of these examples are done in a client/server setting with the server sending the code to be executed on the client machine (and no claim is made that any of these are necessarily agents). Examples of a client sending code to a server machine include the obvious systems where the server exists to do heavy computation for the user and in academic designs of shopping or auction bots that run on the auction server in order to conserve bandwidth and have direct access to data. It is not strictly necessary that one of the machines be under the direct control of the user, but it is generally so. Outside of grid computing, the easiest examples of migrating between multiple machines, none directly controlled by the user, are malware, such as internet worms. (Somewhat unfortunately, most malware makes an excellent example of software agents.)

We use the term "remote agent" to denote an agent that resides on a different machine than it is generally being accessed from. It is distinguished from a mobile agent in that it doesn't migrate between machines.

1.4 Contributions

The primary contributions of this work are:

- A new form of redactable credential, emphasizing speed, and which is efficient

for large credentials with many claims, presented in Chapter 3. Based on Merkle hash-trees, this credential is orders of magnitude faster than related credentials, and it provides limited anonymity with auditability, instead of trying for complete anonymity.

- An entirely new problem area of redactable signatures with disclosure dependencies, complete with a novel threat model and an efficient solution, presented in Chapter 4. Based on a novel encoding of dependencies in a loose hash-graph, this signature scheme is very fast and proven secure even in an open-ended "can prove" security model where both the prover and verifier can conspire together to break the system. This is the first known redactable signature scheme to handle disclosure dependencies. It operates in polynomial time, while a naive solution enumerating all valid data combinations is exponential in time in the worst case.

- A system for providing user-centric identity management services, centered around the storage and usage of long term credentials, presented in Chapter 5. This system, named GUIDE-ME, uses the Merkle hash-tree credentials as part of a protocol to provide secure authentication between entities who are may have never communicated before.

- A framework for a generalized, user-centric identity agent, along with a number of applications for it, presented in Chapter 6.2. This extensible framework, named GUCIdA, includes protocols and credential handling inherited from GUIDE-ME, combined with additional support for communications, database storage, and other features.

- The Medvault system for the patient-centric and source-verified storage of medical records with attribute-based access control, presented in Section 6.4. The current implementation of Medvault is built on the GUCIdA framework.

1.5 Dissertation Organization

The remainder of this thesis is organized as follows:

- Chapter 2 reviews previous work on digital credentials, authentication services, and agents and related distributed systems.
- Chapter 3 presents the first credential design, based on Merkle hash-trees, including security analysis and performance results
- Chapter 4 presents the second credential design, based on looser hash-graphs and designed to cryptographically enforce dependencies between claims. The system's security and performance are discussed.
- Chapter 5 presents a system for authentication based around a network resident, credential-holding agent.
- Chapter 6 presents a generalized framework for building remote agents and other software entities, along with the main demonstration application, a system for user-controlled access to medical records.
- Chapter 7 provides concluding remarks and discussion.
- Appendix A provides more formal proofs of security for the credentials presented in Chapter 4.
- Appendix B provides an outline of a security proof for the protocols used in Chapter 5.
- Appendix C contains a list of papers produced in the course of this research.

CHAPTER II

RELATED WORK

2.1 Digital Credentials

Digital credentials are an important topic in identity management. Credentials are "evidence of authority, status, rights, entitlement to privileges, or the like" [47]. Digital credentials are therefore digital evidence, by which is meant purely information, with no specific physical embodiment. With respect to intellectual property, Stefan Brands treats the term "Digital Credentials" as referring solely to his system [7]. The term was briefly trademarked by a holding company [61], but is currently (July 2009) abandoned.

2.1.1 Chaum's Credentials

David Chaum is credited with proposing the first digital credential system[17]. Chaum's system is a pseudonym system, whereby a user can present a different pseudonym to each service provider. The pseudonyms are linked cryptographically such that even colluding service providers cannot link together two pseudonyms belonging to a user, but the user can use a credential assigned to one pseudonym with another pseudonym. Chaum's original proposal did not describe how to implement such a system.

A method for implementing the system was later described by Chaum and Evertse based on the RSA crypto-system and a semi-trusted third party [18]. Their solution is to use a single RSA modulus N, whose factorization is only known by the semi-trusted third party, known as the signature authority. The signature authority is known to "organizations", which are entities that issue credentials, but does not have to be trusted by "individuals", entities that use, but do not issue, credentials.

Simple version: Setup: Signature authority generates p, q, $\phi = (p-1)(q-1)$,

$N = p * q$ Also generates a list C of possible credentials (each c in C is a positive integer co-prime with ϕ), and $b = sum(C)$. N, C, and b are public. p, q, and ϕ are private.

Each individual gets a primary identity u, which is a random element of Z_n^*, chosen by the signature authority.

Pseudonyms: An individual creates a pseudonym for an organization by simply generating a random element of Z_n^*, r, and then computing pseudonym $= u * r^b \pmod N$. A different r value is used with each organization, leading to different pseudonyms. For example the individual's pseudonym with organization A is denoted as $u_A = u * r_A^b \pmod N$ and the individual's pseudonym with organization B is denoted as $u_B = u * r_B^b \pmod N$.

Issuing a credential to a pseudonym: An organization A issues a credential c (which is an element of C) to an individual's pseudonym u_A by asking the signature authority to compute $d_A = u_A^{c'} \pmod N$, where $c' * c = 1 \pmod \phi$. In normal RSA notation, the signature authority signs the message u_A using the private exponent $(d = c')$, corresponding to the public exponent $e = c$.

Showing a credential on a different pseudonym: The individual can then show the credential d_A, c to an organization B by computing $d_B = u_B^{c'} \pmod N = d_A/(r_A^{(b/c)}) * (r_B^{(b/C)}) \pmod N$. Organization B verifies the credential d_B, c on pseudonym u_B by verifying $d_B^c = u_B \pmod N$.

Complication: In the simple form described so far, users can cheat by creating their organizational pseudonyms in a way that credentials issued to one individual may be transfered to another individual. So, an additional layer of validation must be applied.

Additional setup: The signature authority needs to generate a set of validators. Each validator is a pair of primes p_i, q_i, and one validator is used for each organization. Additionally, the signature authority generates an extra value a and a set m of

elements in Z_n^*. The size of m is a security parameter, where the chance of cheating is $1/(|m|$ choose $|m|/2)$.

2.1.2 Brand's Credential

Stefan Brands developed a form of digital credential in which a user has a single public credential, but that credential is pseudo-anonymous, even to the issuer [8]. The credential holds attributes that the user can selectively prove to a service provider. Repeated showings of the same credential are linkable, however. Since the credential is issued blind by the identity provider, effectively a user has one global pseudonym. The credential can be reissued easily, allowing the user to change the global pseudonym, as permitted by the identity provider. Brands emphasizes that showing a credential is done by zero-knowledge proof, but that has no impact on the comparisons in this thesis. Credentica's U-Prove Software Development Kit is based on Brands' work [9].

Brands' credential can be seen as an extension of Chaum and Pedersen's zero-knowledge signature scheme to many, connected base-exponent pairs. This is referred to as the "relaxed discrete log" problem in [19], where a solution is presented using many rounds of a coin-flipping style zero-knowledge proof.

For the basic version of Brands' scheme, consider a public group G_q, public generators $g_0...g_n$, private attributes $x_0...x_n$, and public element $h = g_0^{x_0} * ... * g_n^{x_n}$ under group G_q. (Brands refers to g_0 as h_0 and x_0 as α, with α being the secret key of the credential.)

The protocol for proving ownership of the credential is as follows:

The prover generates a random set $w_0...w_n \epsilon_R Z_q$ and computes witness $a = g_0^{w_0} * ... * g_n^{w_n}$. The prover sends h and a to the verifier, along with a signature on h (method unspecified for now).

The verifier generates a random challenge $c \ \epsilon_R Z_q$ and sends it to the prover.

The prover computes $r_i = c * x_i + w_i \pmod q$ for i in [0..n], and sends all of the

r_i to the verifier.

The verifier checks that $h^c * a = g_0^{r_0} * ... * g_n^{r_n}$

It is easy to see that $h^c * a =$

$(g_0^{x_0} * ... * g_n^{x_n})^c * (g_0^{w_0} * ... * g_n^{w_n}) =$

$(g_0^{c*x_0} * ... * g_n^{c*x_n}) * (g_0^{w_0} * ... * g_n^{w_n}) =$

$g_0^{c*x_0+w_0} * ...g_n^{c*x_n+w_n} =$

$g_0^{r_0} * ... * g_n^{r_n}$

There is another property Brands' emphasizes: the ability to issue a credential that is anonymous with respect to the issuer.

2.1.3 Camenisch's Credentials

Camenisch, et al., have proposed and implemented yet another form of digital credential, or more precisely, yet more forms [11, 12, 13]. While they describe a system for implementing a Chaum-like pseudonym system, their system is much more flexible, and can be used without pseudonyms. While having significantly better anonymity properties, the algorithms are also significantly slower than Brands' credentials. IBM's idemix system is based on Camenisch, et al.'s work [11].

The pseudonym version is as follows: Each organization has a public key PK_O, an RSA modulus N_O, and five elements a_O, b_O, d_O, g_O, h_O. Each user has a private key x_U. The pseudonym of a user U at a site O is denoted $N_{(U,O)}$ in [13]. The value has no actual use in the protocol, however, and presumably exists only for the convenience of reference. (For example, $N_{(U,O)}$ could be "john.doe@domain.com".) The actual pseudonym of a user, from the protocol point of view, is a tag $P_{(U,O)}$, such that $P_{(U,O)} = a_O^{x_U} * b_O^{s_{(U,O)}}$ where $s_{(U,O)}$ is a random string generated jointly by the user and organization, but known only to the user. A credential issued by an organization to a user is a pair $(C_{(U,O)}, E_{(U,O)})$ such that $C_{(U,O)}^{E_{(U,O)}} = P_{(U_O)} * d_O \pmod{N_O}$. Credentials are shown via zero-knowledge proof, such that the verifier

only learns that the organization issued a credential to the user, and not the value of that credential. This limits organizations to issuing effectively binary credentials, unless they issue multiple sets of public parameters, instead of just a single set per organization. The other versions of the credentials are based on slightly different problems, but all follow the same zero-knowledge proof idea.

2.1.4 Johnson's Homomorphic Signatures

The closest work (in terms of mechanism) to the minimal disclosure credentials presented in a later chapter is the redactable signature scheme by Johnson et. al. from their Homomorphic Signature paper [31]. They describe not a credential, but just a simple cryptographic signature. However, the mechanism of their signature, using Merkle hash trees, is the same underlying mechanism as my system, described in a later chapter.

Due to the similarities, Johnson et. al's redactable signature will be described later in Section 3.1.4, when it can be contrasted with the minimal disclosure credentials.

2.2 Authentication Services

Many systems have been designed for holding and exchanging credentials for identification and authentication. These systems can generally be divided into systems that just hold credentials and systems that provide some form of federated authorization. An example of a system that just holds credentials is Microsoft Passport. Examples of systems that provide some form of federated authorization are OpenID and Shibboleth.

2.2.1 OpenID

OpenID is an open standard for authorization based on proving ownership of a URL. [50] OpenID is most associated with blogs and other user-generated content

sites, as it was originally developed by the creator of the LiveJournal site [23]. However, while associated with low-security applications, the underlying protocols are comparable in security to other single-sign on systems.

The architecture of OpenID is shown in Figure 1, with communications links. [1] (Note that there are two variants of the protocol, distinguished by whether the identity provider keeps session state or not. Figure 1 covers both variants.) In OpenID, the user's agent is an unmodified web browser, so the only function required of it is the ability to pass messages via redirection. Similarly, the lookup service (which controls the URL the user is claims control over) can be an unmodified web server, as it only has to serve plain files to the relying party. The identity provider and the relying party must both be aware of the protocol, and carry out all cryptographic operations.

The OpenID protocol can be simplified to the following:

1. The user identifies them self to the relying party as owning a certain URL.
2. The relying party goes to that URL (which is at the lookup service), and gets the contact information for the identify provider that handles that URL.
3. The relying party contacts the identity provider and establishes a shared secret.
4. The relying party tells the user to ask the identity provider for the shared secret.
5. The user authenticates to the identity provider and gets the shared secret.
6. The user presents the shared secret to the relying party.

2.2.2 Shibboleth

Shibboleth is an open source software package for federated authentication, built on SAML (Security Assertion Markup Language). Shibboleth is most associated

[1] The terminology of this figure matches the usage in this dissertation; the OpenID specification uses different terminology.

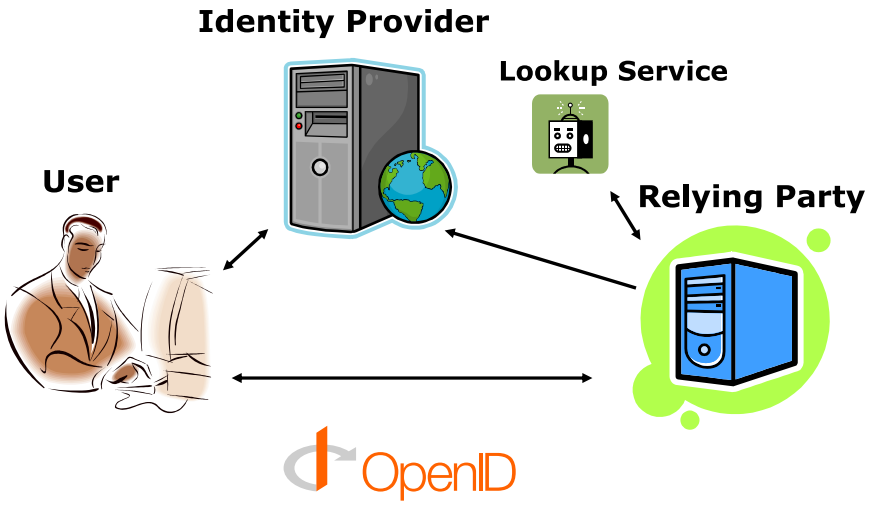


Figure 1: OpenID Communications Architecture.

with authentication within and between universities, as per its academic origin [57]. Figure 2 shows the communications architecture for the Shibboleth, version 1.x.

Shibboleth refers to the lookup service as the Where Are You From (WAYF) service, and divides the identity provider and relying party into multiple pieces. For example, the identity provider is split into a handle server and an attribute authority.

The Shibboleth 1.x protocol can be simplified to the following:

1. When the user tries to access a resource at the relying party, they are directed to the Where Are You From (WAYF) service.

2. The user identifies their identity provider (home federation) to the WAYF service and are redirected there.

3. The user logs authenticates to the identity provider and gets a handle.

4. The user gives the handle to the relying party.

5. The relying party presents the handle to the identity provider.

6. The identity provider gives the relying party the requested information about the user.

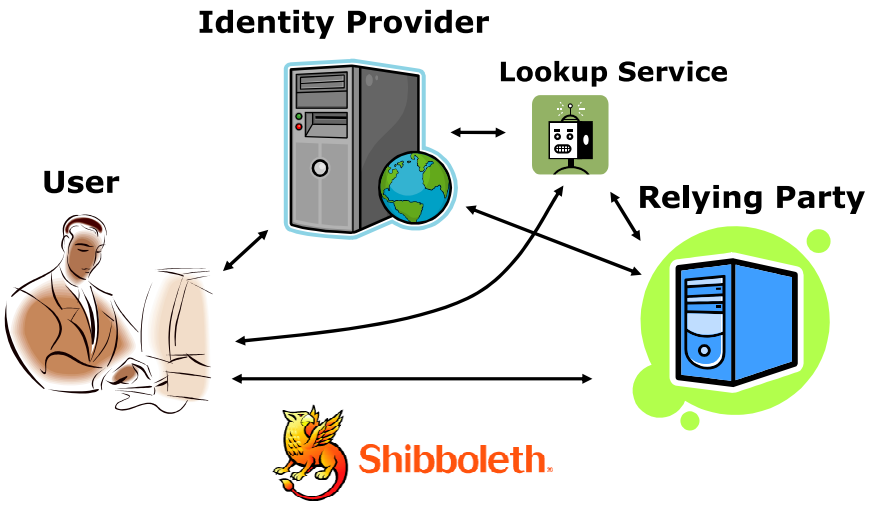


Figure 2: Shibboleth Communications Architecture.

CHAPTER III

MHT BASED CREDENTIALS

The first part of my research is on the area of credentials and authentication. From this work came two credential designs. Both designs fix the values of claims exactly, so micro-claims should be used with them (as discussed in Section 1.2.3).

3.1 Design

3.1.1 Merkle Hash Trees

The first credential design is based on Merkle hash trees [39] and standard public-key infrastructure (PKI) certificates. A Merkle hash tree is essentially a binary tree where each internal node holds the hash of the concatenated values of its two children nodes. The leaf nodes hold the data of interest. In this way, a large number of separate data can be tied to a single hash value. In addition, by storing the internal node values (or a subset thereof), it is possible to verify that any of the leaf nodes is part of the tree without revealing any of the other data. Ralph Merkle first introduced this structure as a way to efficiently handle a large number of Lamport one-time signatures. It has since been adapted for uses such as the large-scale time-stamping of documents [2] and tracking data in peer-to-peer networks [15]. A basic Merkle tree is shown in Figure 3.

Figure 4 shows the same tree with the nodes and claims labeled for easy reference. Consider verifying claim A in the tree. Starting at the claim and going up the tree, node 4 contains the hash of the claim. Node 3 contains the hash of the concatenation of nodes 4 and 5. Node 2 contains the hash of the concatenation of nodes 3 and 6. And, finally, node 1, the root of the tree, contains the hash of the concatenation of nodes 2 and 9. Therefore, verifying claim A requires the values of nodes 5, 6, and 9,

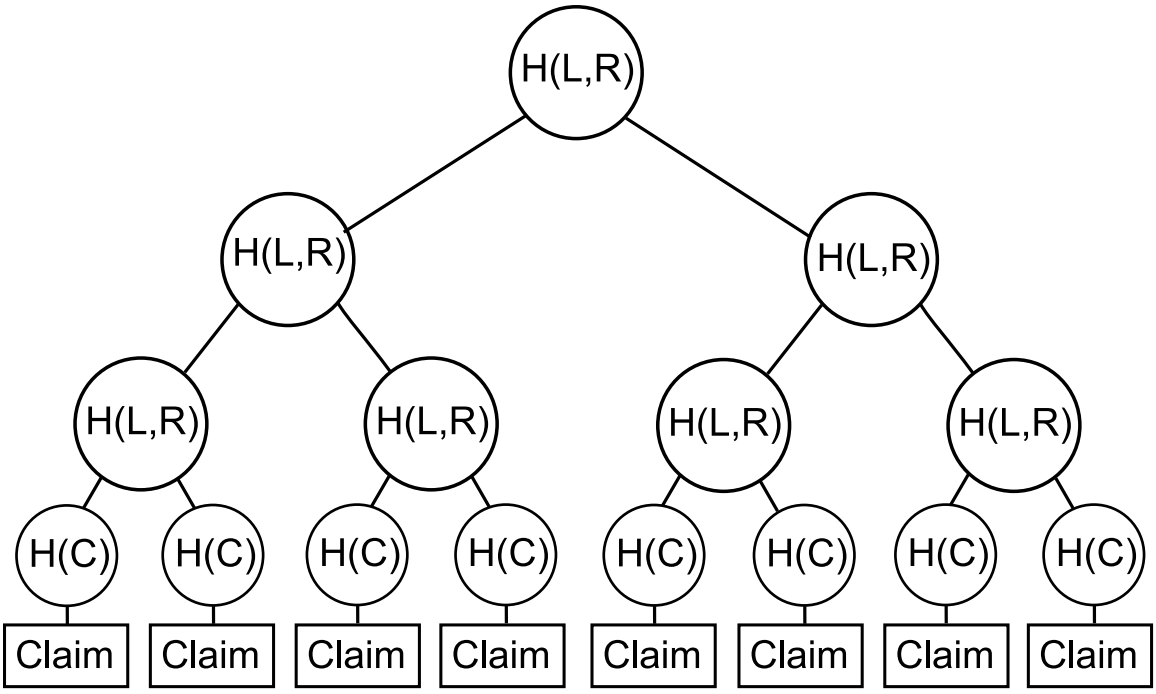


Figure 3: Merkle hash tree with leaf nodes holding hashes of claims.

as shown in Equation 1, below. A similar verification path can be made for any of the claims. For example, Equation 2 shows the path for verifying claim F, which requires the values of nodes 2, 11, and 13. As per the nature of the binary tree, the number of nodes and hashes needed for a verification scales with the log of the number of claims.

$$H(H(H(H(A), 5), 6), 9) \tag{1}$$

$$H(2, H(H(11, H(F)), 13)) \tag{2}$$

3.1.2 Credential overview

The credential consists of two parts: a public part and a private part. The public part of the credential is a certificate. The certificate holds information about the issuer, the certification chain for the issuer, the type of certificate, the date range over which the certificate is valid, the user's public key, and the root hash of a Merkle hash tree. The certificate should not hold any data about the user directly, even data

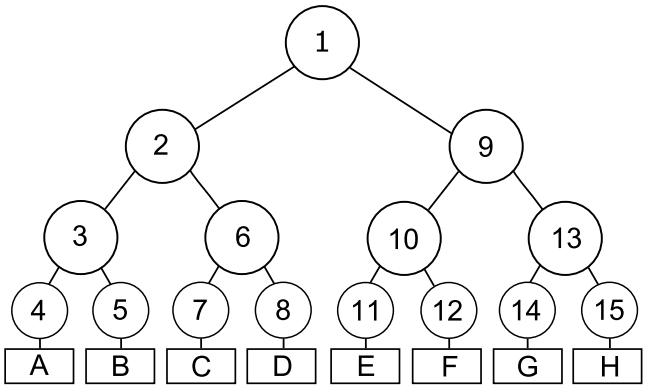


Figure 4: Merkle hash tree with labeled nodes.

as common as a name. As per standard operation, the certificate will be signed by some certificate authority, which is an identity provider in this system. The private part of the credential consists of a private key and a Merkle hash tree whereby all of the leaf nodes are (hashes of) micro-claims about the identity of the user, who is the credential holder. The Merkle tree structure allows the credential holder to prove any subset of the claims in the tree, with only the single signature on the certificate.

As an additional improvement, a slight modification to the structure will allow the use of a single credential containing claims from a variety of identity providers, without requiring one identity provider to verify all of the claims. Consider again the Merkle tree of claims, but with subtrees coming from different identity providers. For example, one subtree could contain claims certified by a government registry, while another subtree could contain claims certified by an employer. In the basic structure, the identity provider must either see all of the claims or trust the providers of all of the subtrees that they only contain claims relevant to their topic. Neither solution is ideal. To provide a third option, consider adding an optional third branch to some internal nodes of the full tree. (These nodes correspond to root nodes of the subtrees.) The third branch contains a certificate for all of the claims in the subtree rooted in the parent node. When verifying a claim which is in such a subtree, the form is different—three hashes are concatenated in a node instead of two—so the

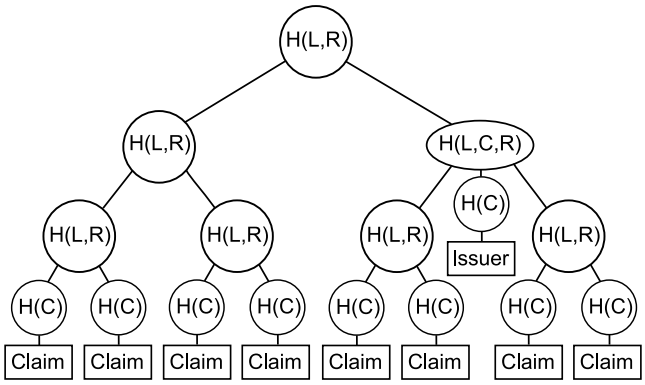


Figure 5: Modified Merkle hash tree with subtree.

verifier knows that this claim is under a different certificate than the credential as a whole. For example, in Figure 5 the left hand subtree contains claims certified by the overall identity provider. The overall identity provider is referred to as the certificate authority (CA). The right hand subtree contains claims certified by some other party, which do not have to be verified by the top level CA. From here on, the term "subtree" is used to refer to a branch containing nodes from a different identity provider, rooted in one of these sub-roots.

We would like to identify a node as being a leaf node without seeing the associated claim. This is required for the security proof in the appendix. This discrimination is achieved by simply appending a bit to the end of the hash, either a one if the node is an inner node, or a zero if the node is a leaf node. For efficiency, our implementation simply overwrites the least significant bit of the hash value, instead of appending another bit. This is equivalent to truncating the output of the hash function used by a single bit (with a insignificant decrease in security), and then appending the indicator bit. In order to keep the system secure while still only needing to check the signature on a single certificate, nested subtrees are forbidden. If a service provider encounters a claim in a nested subtree, it should reject the credential.

A tree might have no claims outside of subtrees. In this case, the top level CA is signing that it has verified that the form of tree is correct, that all certificates for

subtrees are valid, and that the user possess the private key(s) matching the subtree certificates and the top level certificate. The CA could be fully automated in this situation, allowing users to easily update their credentials when subtrees are added or modified. Note also that changes to attributes in one sub-tree only require recomputation of that subtree, the main root hash, and the top-level signature. Recomputation of other sub-trees and other signatures is *not* required.

3.1.3 Protocols for the Credential

Creating a credential is both conceptually and computationally easy. In the case where there is a single identity provider for the credential, there are roughly four steps required: first, agree on a list of claims. Next, generate the hash tree for the claims. Third, verify that the user possesses the private key. Finally, produce and sign the public certificate.

First, the user and identity provider agree on a list of claims. The logistics involved in an identity provider verifying the user claims are beyond the scope of this paper. Second, either the user or the identity provider generates the hash tree for the claims. Random padding must be added to the claims before they are hashed, as discussed shortly. With a single identity provider, the tree will always be balanced, bounding the number of interior nodes to the number of claims. The number of hashes needed to generate the tree is therefore bounded to twice the number of claims—generally an insignificant amount of computation time. Third, the identity provider must verify that the user holds the private key matching the public key of the credential. The user can reuse an already generated key pair or generate a new key pair for the credential. The identity provider does not need to ever know the private key. Finally, the identity provider creates and signs the public certificate for the credential.

Random padding must be added to the claims before they are hashed in order to prevent dictionary attacks against parts of the tree. The random padding can be

generated and stored in a number of different ways. For example, Johnson, et al. [31], describe a method credited to Goldreich, Goldwasser, and Micali [25] that uses a pseudo-random function whose output is twice the length of its input. A single seed value at the root node is expanded as per the tree branching structure using the pseudo-random function. Each node therefore has a pseudo-random value associated with it that can be used to generate the pseudo-random values for all of its children nodes. A computationally easier method is to use a seed value and the claim index with a pseudo-random function to generate the padding for each claim. Neither of these methods will work well for our system, unfortunately, due to the inclusion of different subtrees within a larger tree. In the most general case, simply storing the random padding for each claim is probably the easiest solution. The overhead is small, just 10-16 bytes per claim. Alternative approaches are possible, but all require the user to have a global secret, keep the private keys for all certificates, or store extra private information with each credential.

Creating a credential with claims from multiple identity providers is done by integrating credentials as subtrees. One identity provider will be the final one, referred to as the certificate authority (CA). The user creates credentials with all of the identity providers, except for the final one, by the procedure described above. Then, the user creates a credential with the CA. For this final credential, the hash trees from the other credentials are incorporated into the final hash tree. The root nodes for the incorporated subtrees will have a third branch added, each containing the certificate from the original identity provider of that subtree, as shown in Figure 5. (The root hash values in the subtree certificates will no longer match, but that doesn't matter.) All subtrees in the credentials being added must be removed from their trees and added separately to the top level tree. This prevents nested subtrees. The CA verifies the structure of the tree, including the top two leaf nodes of each subtree (as they

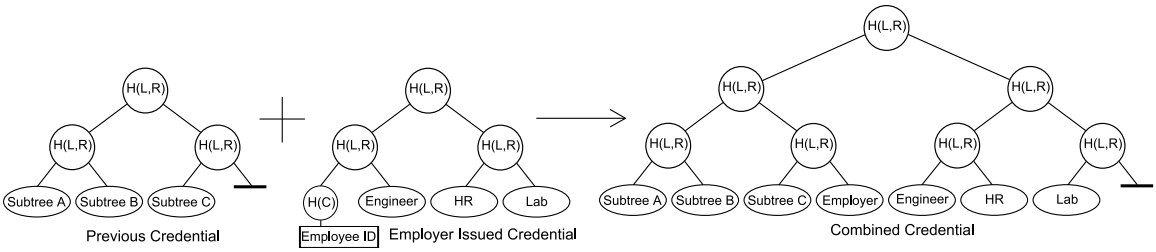


Figure 6: Combining a generic credential with an employee credential.

are needed to calculate the sub-root's hash), as well as the public part of the sub-credentials, and the associated public/private key pairs. In general, the public/private key pairs may be the same as or different from each other and the top level key pair. All that matters is that the user holds the corresponding private keys at the time that the credential is assembled. In a more restricted setting, the public/private key pair may be required to be the same for all subtrees. The CA does not need to see the claims from subtrees.

To provide an example of combining credentials, imagine a user, Alice, who already has a credential containing claims from three different identity providers. Alice is then issued a credential from her employer. The two credentials are shown in Figure 6. Alice's employee credential already has a number of subtrees, containing separate credentials from the engineering department, human resources department, and her own lab. The credential also has a top level claim of her employee ID. Alice goes to a certificate authority to create a new credential from these two credentials. She submits no claims to be verified by the CA and seven sub-credentials to be included in the final credential—three of these are from her previous credential, three were embedded in her employee credential, and the last is the credential claiming her employee ID. The CA will verify the seven certificates, verify that Alice possesses the appropriate private key(s), generate a new tree, and provide Alice with the new certificate.

The protocol for using the credential follows conventional PKI certificate usage.

The user connects to a service provider, either over a wide or local area network, and requests some service, sending the public part of the credential. The service provider requests the appropriate identity attributes from the user. The user provides the claims that match the requested attributes and the intermediate node values and path information necessary for the service provider to verify each of the claims. If any of the claims are in a subtree certified by a different identity provider, the accompanying certificates must be included with the set of claims. The service provider verifies that the claims are in the hash tree specified (via the root hash) in the certificate part of the credential. The service provider also verifies the signature on the certificate part of the credential. In order to verify that the user is the holder of the credential, the service provider also verifies that the user possesses the private key which matches the public key claimed by the credential. This can be done by standard methods, such as challenge/response or as part of a secret key agreement operation, as long as the key verification is tied to the specific claims being asserted by the user. This can be done, for example, by hashing all of the claims asserted by the user as part of the challenge field. In addition to the cryptographic verifications, the service provider must of course confirm that it trusts the identity providers to assert the claims in the credential. For example, a claim of an individual's address asserted by the Bar Association would be out of place. Similarly, an assertion by the Department of Motor Vehicles that an individual was a licensed lawyer should not be trusted.

The procedures for creating and using credentials given herein are purposely generic. We prefer to constrain the structure and properties of the credential and leave some flexibility in the procedures for generating and verifying it. As a prototype, we have developed specific implementations of these procedures, and performance results based on these implementations are reported in Section 5.

3.1.4 Comparison to Related Systems

Johnson, et al., define and construct redactable signatures using a Merkle hash tree to allow the signature verification of a message even when parts of the message have been deleted [31]. In their example, a body of text is signed such that it can be redacted at some level of granularity (sentence, word, or character making the most sense). They do not appear to have considered using the construct in the context of a credential. While both systems share a core idea—using a Merkle hash tree to hide signed elements—the two systems have significant differences. Both systems use slightly different hash functions (constructs) for the leaf and inner nodes. In their signature scheme, the input to the hash function is specified to match a certain form, which is important to their proof of security. In our credential system, the output of the hash function is specified to match a certain form, which is sufficient to meet the requirements of the security proof, while providing the property of being able to identity a leaf node from its hash value alone. Subtrees of the type used in the credential system are not used at all in the signature scheme. To the best of our knowledge, the idea of combining trees by modifying the tree structure as we do is novel, and introduces extra complications. The three-child inner nodes (shown in Figure 5) and modified hashing scheme (discussed in Section 3.2) both result from the idea of combining subtrees from different identity providers. Combining subtrees allows claims from many different sources to be in a single credential, under a single certificate.

Brands mentions in his book the idea of using hash trees to store claims, but dismisses it because it does not provide the properties he desires [8].

3.1.5 Copy Resistance

The credential is strongly copy resistant under normal operations, due to the incorporated public/private key pair. Whenever the credential is used, the private key is

used to prove that the user is the authorized credential holder. Since the private key never leaves the user's machine, it can't be copied by the service provider or any third party listening to the exchange. (The private key can of course be copied if the user's machine is compromised, but that is beyond the scope of this paper.)

This copy resistance property extends to preventing standard man-in-the-middle attacks. For example, consider how a phishing site handles conventional one-time passwords or other two-factor authentication methods. A user logs in to what they believe is the legitimate site of their bank, broker, or other service provider, but which is really a phishing site. The user enters their user-name and password and then, either in the same step or in a second step, enters a one-time password from a sheet of paper or electronic device. The phishing site simply passes all the information from the user on to the site being spoofed, and returns the responses from the legitimate site to the user. After the user has successfully logged into the legitimate site, through the phishing site, then the phishing site is logged in as the user and can perform whatever actions it wants.

The standard protection against this type of man-in-the-middle attack is the use of server-side certificates with SSL/TLS. When the user initiates a secure connection to a service provider, the service provider replies with a PKI certificate. The user's client (web browser) usually handles checking the validity of the certificate automatically, interrupting the user only when a problem is detected and simply showing a non-intrusive indicator when the secure connection is setup without problems. Phishing sites have used a variety of techniques to get around server-side certificates, including simply not using SSL/TLS, making the browser look like it is using SSL/TLS when it is not, and obtaining valid certificates for different domains, which can be mistaken for the legitimate domain by an unwary user. All of these tricks are possible because of the disconnect between the authentication system (SSL/TLS), and the thing being authenticated (the service provider). The certificates authenticate the domains used,

instead of the actual service providers.

In contrast to the server-side certificates, our credentialing scheme intimately ties the authentication system (the private/public key pair and the certificate as a whole) to what is being authenticated (the credential as a whole). Consider again the phishing site, but with the credential in place of the user-name, password, and one-time password. When the user connects to the phishing site, the phishing site can relay requests to the legitimate service provider and replies back to the user as before. However, when the service provider sets up a secure connection, it will use the user's certificate to setup the session. The phishing site can substitute its own certificate, but then it is no longer impersonating the user. Alternatively, the phishing site can pass on the user's certificate, but then the phishing site loses control of the session. The service provider can setup an authenticated tunnel to the user, so that the phishing site can no longer modify the traffic without being detected.

3.2 Security

In this section we discuss some potential attacks against implementations. We provide a formal analysis of the security properties of the underlying cryptographic construction in the appendix.

3.2.1 General Security Discussion

The basic attacks that the credential must resist are forgery, theft, privacy compromise, man-in-the-middle, and collusion.

Forgery - a malicious user should not be able to forge a valid credential containing invalid claims.

Theft - the credential should be resistant to theft.

Privacy compromise - an attacker should not be able to learn more about the user of a credential than the user chooses to reveal.

Man-in-the-middle - a hostile service provider should not be able to misuse or reuse a user's credential that is presented to it.

Collusion - two or more users working together should not be able to make a claim that no single user in the group can make alone.

During the design process, several specific attacks against earlier versions of our system were discovered. They include dictionary attacks against neighbor nodes, combined replay/man-in-the-middle, hidden subtrees, and the broken hash attack.

Dictionary attacks against neighbor nodes come from the fact that when a credential is used, the hash values of unreleased claims must be provided. Assuming a secure hash function is used, an attacker cannot determine the value of the unreleased claim via cryptanalytic attack. However, since most claims are likely to be in a standard form, a dictionary attack should often be successful. We protect against this type of attack by padding the claims with random or pseudo-random data.

A combined replay/man-in-the-middle attack may be possible when the verification that a user holds the private key for a credential is completely unrelated to the showing of the tree for the credential. In this situation, an attacker can perform a replay attack against the showing of the tree, and then a man-in-the-middle attack against the user proving possession of the private key. While this attack either does not apply or would be meaningless to many uses of the credential, it is still a possible attack. We protect against this type of attack by requiring that the verification of the private key be linked to the specific claims being shown.

To maximize the efficiency of showing a credential, we would like to minimize the number of slow, public-key operations performed. The best that we can do is to have only a single certificate verification and a single public/private key verification, regardless of the number of claims and how many different subtrees the claims may be in. During the issuance of a credential an identity provider will check the certificates for all subtrees within the tree. Therefore, when the credential is shown, the service

provider does not need to recheck those certificates, as long as it trusts the top level CA. However, consider the case of a nested subtree. An identity provider shouldn't be able to see the claims in subtrees, due to privacy requirements. But this means that a subtree can hide another subtree. In particular, a subtree issued by a bad (but untrusted) identity provider could hide a subtree apparently issued by a trusted identity provider, but in actuality was forged and has a bad signature. This could be prevented by several different protocol changes. The most efficient of these is by preventing the inclusion of a hidden subtree in the first place. Our current recommendation is simply to ban nested subtrees altogether, and have the service providers ensure that credentials which have nested subtrees are rejected.

Another structural attack that was encountered is implementation specific. In our prototype implementation, plain X.509 certificates are used. [29] X.509 certificates are designed to match a directory entry for a particular subject, and their structure is fairly rigid around that purpose. Therefore, our implementation didn't specify any extra details about the hash tree—such as the hash algorithm used—in the certificate itself, but instead stored that information with the tree. This can lead to an attack if a hash algorithm trusted for use in creating the tree is broken. Given the right conditions, an attack could create a tree using the broken algorithm, which could then collide against the root hash stored in a valid certificate, even if the valid certificate was for a tree using a different (and hopefully more secure) hash algorithm. We consider this attack to be fairly minor, because it requires a primitive of the system to be broken badly—in order to find a collision from one hash function to another should effectively require a preimage attack, and not a simple collision. As version 3 of X.509 certificates allows arbitrary extensions, the hash algorithm can be specified in the certificate, anyway.

3.3 Performance

Our minimal disclosure credential design focuses on conventional operations (one-way hashes) and minimizes the number of asymmetric/public key operations. This allows our system to be much faster than the digital credential schemes by Brands and Camenisch, et al., which require many more of these expensive public key operations. In this section, we report on two sets of performance experiments that validate this claim. In the first set, we evaluated the computation time to verify credentials of different types with a varying number of claims. This set focuses solely on the computational burdens placed on servers, which is likely to be the most significant performance bottleneck in a system. In the second set of experiments, we implemented and deployed an actual client-server system in Emulab [65], and evaluated the server throughput. This set of experiments included communication costs and additional client-side and server-side computations costs in addition to credential verification time, and it has multiple clients accessing the service simultaneouly, to provide a realistic overall assessment of performance. In both sets of experiments, we compared our hash tree approach against Brands' credentials and the brute force solution of embedding each claim in a separate certificate. For Brands' credentials, we used the parameters from [10, Section 2.2] ($|p| = 1600, |q| = 256, |s| = 160$).

The first set of experiments (credential verification time only) was performed on an Intel Core 2 Duo E6600 running at 2.4 GHz using Sun's server JVM version 1.6.0. In all cases the operations were run repeatedly before timing, so as to force the JVM to fully optimize the operations. Our prototype uses X.509 certificates (with the root hash as the distinguished name) for the public key certificate part of the credential. The certificate is signed using RSA, and the CA's key is 1536 bits. Results are shown in Table 1. The total time to verify one of our hash tree credentials is equal to the tree verification time plus the time to verify one certificate. Trees containing claims from multiple authorities would require verifying the hash tree and one certificate for

Table 1: Time efficiency of hash tree and certificate

Operation	Time (μ-sec)
Verify Tree (SHA-256)	
1 claim of 2048	31
20 claims of 2048	100
All 2048 claims	8030
Verify Certificate	
RSA, 1536 bit	620
Verify Brands	
1 claim	38,000
20 claims	283,000
2048 claims	26,000,000

each authority that has at least one claim being asserted. The hash tree used in these experiments contains 2048 claims. From Table 1, we see that verifying one claim with the hash tree approach is approximately 58 times faster than verifying one claim in a Brands' credential. For 20 claims, the hash tree approach is approximately 390 times faster than Brands', and with 2048 claims, the speedup factor is approximately 3,000. Speedup compared to the brute force approach of one certificate per claim is also substantial. This speedup is linear with the number of claims, i.e. approximately 20 times faster with 20 claims, 100 times faster with 100 claims, etc.

Our second set of experiments was done using an actual client-server implementation deployed in Emulab [65]. In these experiments, there were a variable number of clients and one server, which accepted connections from the clients and verified one credential per client request. Each client and server instance was run on a separate 3 GHz Pentium Xeon. Each client generated verification requests one after another, waiting for one response and then immediately generating a new request. We again waited until the server had processed a number of requests before timing the operations to ensure full JVM optimization. All parameters were the same as in the first set of experiments. Here, our primary performance measure is the server throughput (number of verifications done per second). The results of these experiments are shown

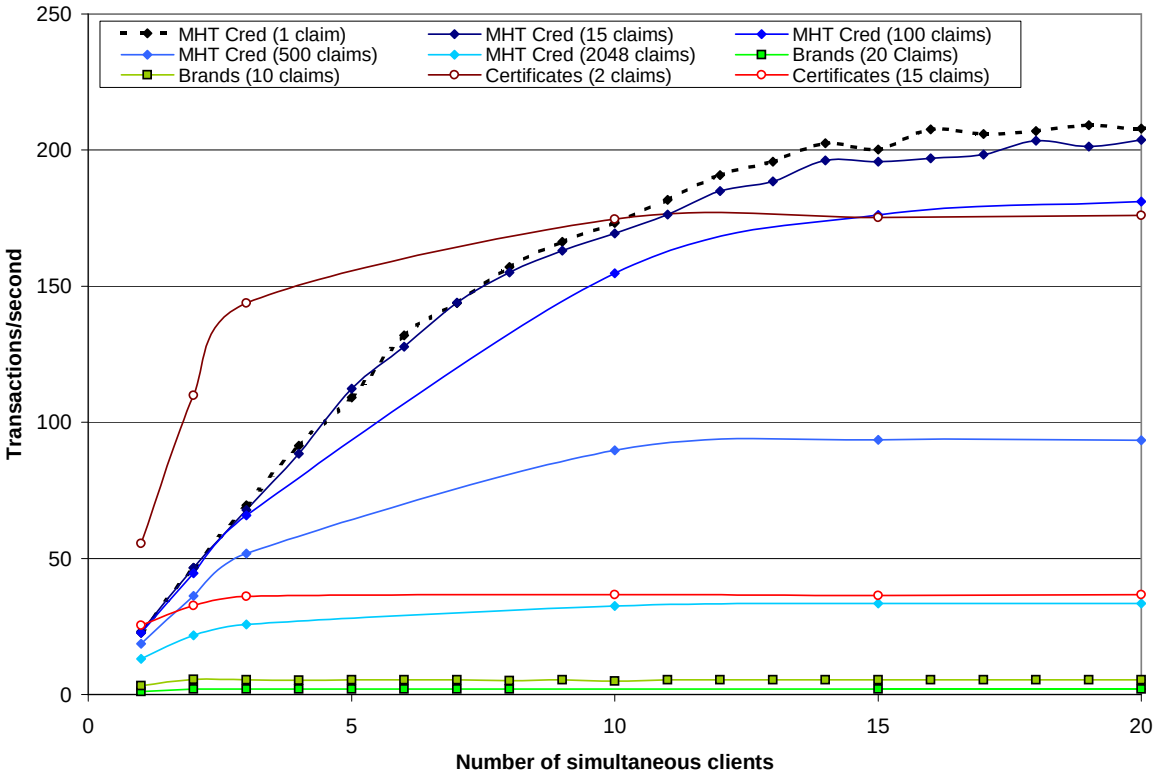


Figure 7: Credential Verification Throughput vs. Number of Clients

in Figure 7. The figure shows that throughput with Brands' credentials saturates at only a few clients, while the hash tree credentials can handle up to about 15 clients before saturating. The peak throughput for the hash tree credentials is approximately 210 verifications per second, while the peak throughput with Brands' credential having 10 claims is approximately 5 operations per second and with 20 claims it is only about 2 operations per second. Adding claims to the hash tree credential has only a modest impact on performance. With 15 claims, the peak throughput is still over 200 verifications per second and even with 2,048 claims, the throughput is still around 33 verifications per second. The figure also shows that throughput for the brute force approach with one certificate per claim drops off rapidly as the number of claims increases. Overall, the Emulab results support the earlier experiments in that the hash tree credentials retain a strong performance advantage relative to Brands' (approximately 40-100 times faster for moderate numbers of claims).

3.4 Additional Variations

3.4.1 Strands implementation

The design presented in this chapter follows the view of starting with a large credential and masking off the parts that are not being shown. A partially alternative implementation [1] was written to allow partial trees to be built up from pieces, instead of stripped down from full trees. Each piece (called a "strand") consists of a single claim and the hash values needed to verify that claim. The needed hash values are exactly the same as if the full tree is reduced to a partial tree with just the one claim. But the strand implementation is more space efficient and allows strands to be combined back together to form partial or complete trees. The strand implementation is used in Medvault, described in Section 6.4.

3.4.2 Zero-knowledge proof signature

An enhancement to the system allows the credential to be shown to the relying party, but prevents the relying party from proving that the credential is valid to another entity. Instead of storing the root hash in the certificate, the CA provides to the user a trapdoor to perform a zero-knowledge proof that the certificate goes with the root hash. The user can use the trapdoor to prove to the relying party that the certificate goes with the root hash, and therefore the full hash tree and claims in it. The relying party, lacking the trapdoor, cannot make the same proof to another party. This variation is not implemented.

As an example, the zero-knowledge proof connection described above can be done using the Guillou-Quisquater (GQ) identification scheme [38]. The GQ identification scheme uses an RSA-like construction, the details of which are shown in Figure 3.4.2. The scheme is designed to tie a user's identity to a group, determined by some public

[1] The implementation is only partially alternative because it requires the primary implementation for the creation of the hash trees and some other operations.

n = standard public RSA modulus (p and q are secret, known only to the CA)
v = public exponent
$s = v^{-1} \pmod{\phi}$ (CA's secret key)
$j = f(\text{identity})$, where f is a redundancy function
$A = j^{-s} \pmod{n}$ (user's secret key; computed by CA)
$r \in [1, n)$ (commitment)
$x = r^v \pmod{n}$ (witness)
$x \rightarrow$
$\leftarrow e,\ e \in [1, v]$ (challenge)
$y = r * A^e \pmod{n}$ (proof of knowledge)
$y \rightarrow$

$$
\begin{aligned}
z &= j^e * y^v \pmod{n} \\
&= j^e * (r * A^e)^v \pmod{n} \\
&= j^e * r^v * j^{(-(v^{-1})*v*e)} \pmod{n} \\
&= j^e * r^v * j^{-e} \pmod{n} \\
&= r^v \\
&= x
\end{aligned}
$$

Verifies $z = x$, and $z \neq 0$ (ie, $r \neq 0$)

Figure 8: The Guillou-Quisquater identification scheme

parameters. Some modification is necessary to tie a specific identity (a specific root hash) to a specific certificate, without requiring a new set of parameters for each credential. (Generating a new set of parameters is equivalent to generating a new RSA keypair - a far more expensive operation than simply signing the certificate.) In the figure, there is a redundancy function "f" which is not specified. Specific examples of redundancy functions used are PKCS #1 [32] and OAEP [4]. The version of OAEP defined in [32] has an extra parameter, an optional label, which will do exactly what is needed. A random value can be generated for the label, which is then stored in the certificate. When a credential is being verified, the label in the certificate is used to verify that not only has the hash tree in question been signed by the CA, but that it goes with the specific certificate it is presented with. (If the label parameter were not available, the random seed values used in both PKCS and OAEP could be used in the same manner.)

CHAPTER IV

CREDENTIALS WITH DEPENDENCIES

4.0.3 Motivation for Dependencies

From a basic credential system, we expanded the scope of our uses for the credential construct to hold arbitrary records. In doing so, the issue of dependencies between different data items or between different whole records arose. Data seldom exists in isolation. Individual pieces of data are combined to form data, which are combined to form datasets. Data can be connected via many relationships, and releasing individual pieces of data ignorant of those relationships often makes no sense. We wish to provide a way for the certifier of data to prevent data from being disclosed without respecting the relationships between different pieces of data. As mentioned previously, "disclosed" refers only to releasing the data in a certified form. In this work, relationships between data are reduced to dependencies between data items that must be satisfied for the data to be disclosed.

4.1 Related Work

While we are not aware of any other attempts to address this specific problem of dependencies in releasing certified data, this work is still related to various prior works.

The most closely related works using graphs or circuits of dependencies involve fulfilling dependencies under a significantly different threat model, for example [26]. In our system, the most difficult threat is the prover and verifier collaborating to cheat the certifier. And the threshold of that cheating is low, since the result only has to be trusted by the verifier, and not by any honest party.

This work can be considered the core of a specialized redactable signature scheme,

although it is meant to be embedded into a separate redactable signature of the hash-tree type described in the previous section. Redactable signatures were introduced by Johnson, et al., in [31] as one example of a larger class of homomorphic signatures. While the targeted use of the schemes is very different—Johnson, et al., describe a scenario where the majority of a document is shown, with a small part redacted, while our work describes showing a small amount of data and redacting the majority—the core mechanism is the same.

Privacy preserving trust negotiations associated with the disclosure of sensitive attributes also have disclosure dependencies among attributes [67, 58, 56, 68]. A user can set up policies where the other negotiating party has to disclose some attributes before a particular attribute of the user can be released. One major difference between trust negotiation systems and our proposed system is that the trust negotiation systems are online approaches, i.e. they expect the other party to release credentials online to satisfy the dependencies as opposed to our offline system where the dependencies are enforced cryptographically. Another difference is that in trust negotiation systems, both parties try to satisfy each other's policies at run time, whereas in our system the policies result from natural dependencies in the data set that must be preserved to release the information in the correct context. Finally, in trust negotiation, each party trusts its negotiator to faithfully execute its own policies, whereas the party releasing information in our approach is not fully trusted by the source of the information and the releasing party cannot therefore be relied upon to faithfully execute the source's policies.

Other systems attempt to limit the disclosure of personal information/credentials in different ways. For example, in role-based access control, only a user's role needs to be disclosed and not a more specific identifier [53]. Alternately, by adding context to role-based access control, policies for access can be tightened and made more

fine grained, thereby reducing the amount of unnecessary personal information retrieved [28]. In a different use of context information, [52] uses context information (about both the subject and the querier) to determine what information can be released for any request. These works do not consider the issue of verifiability of the data being provided nor its inherent data dependencies.

Our system provides a mechanism for efficiently handling very large systems with arbitrarily complex dependency rules. It does not address how to choose what to show to satisfy the dependencies. We hope that an automated system using pre-selected policy rules can intelligently choose the best disclosure options, or at least provide a user with guidance to make the decision much easier. For example, if this system is used as part of a credential system, automated trust negotiation can determine what options are chosen.

4.2 System Description

Our redactable signature with dependencies consists of several parts. The first two parts are the PKI certificate and Merkle hash tree as described in Chapter 3. The interesting and novel part described in this chapter is the handling of the dependencies. For simplicity, this chapter will largely neglect the PKI and other, higher-level credential details, and just focus on the dependency handling itself.

4.2.1 Dependency Graph

Dependencies between data can come in many forms. The simplest form is a single "depends upon" relationship, such as "claim 1 depends upon claim 2", which means that "claim 1" should not be released without also releasing "claim 2". The next simplest form is a chaining of dependencies, such as "claim 1 depends upon claim 2, and claim 2 depends upon claim 3". These chains can be handled by creating one node per claim in the chain, with each node containing its corresponding claim and all subsequent claims in the chain. Less simple is when there are OR options, such

$\rightarrow$ is used for depends upon
$A \rightarrow B$ is read "A depends upon B",
and A is called a parent of B
+ indicates concatenation
{} indicates a set of values, concatenated together
$S(x)$ is the string for vertex x, and is defined as
$S(x) = H(\{$ parent vertices strings $\} + x)$

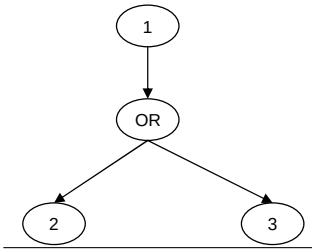
Figure 9: Generic Dependency Form

as "claim 1 depends upon either claim 2 or claim 3". In small numbers, these OR options can be handled by just enumerating the possible combinations as if they were chains, but for large systems, that is extremely inefficient.

We represent simple OR dependencies by a directed acyclic graph (DAG). To handle these dependencies efficiently, a secure hash function is used to create a path whereby a claim is proven valid at the same time as the next node in the graph is proven valid. We call a node that is dependent upon another node a parent of the latter node. The node that is depended upon is called the child node. A node is assigned a "string" value, which is a hash of the string values of its parent nodes and its actual data value. Calculating a node's string therefore requires having the data for that node. Just as the node (hash) values in the Merkle hash tree define a unique set of children, each node's string value defines a unique set of parent nodes and its data value. In order to tie the entire DAG down to a single value, an output value is created, which is simply the set of string values of all of the leaf nodes of the DAG.

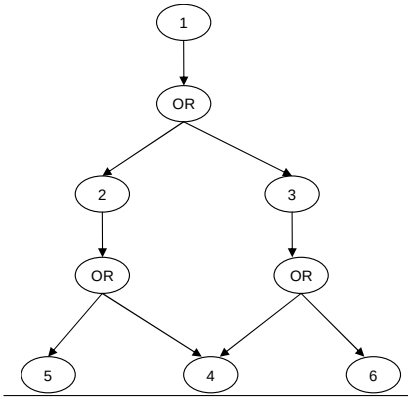
Figure 9 shows the notation that we use, while Figure 10 shows the example described above. The example in Figure 11 shows that multiple parent nodes are efficiently handled. The hashed value of the node for Claim 4 simply has two parent strings instead one. In general, the size of the node's value will grow sub-linearly with the number of parents, as the extra parent strings (each just a single hash value) are amortized by the node's actual data content.

AND operations are more complex to handle than OR operations. An example



$Claim1 \rightarrow Claim2$ or $Claim3$
$S(Claim1) = H(Claim1)$
$S(Claim2) = H(S(Claim1) + Claim2)$
$S(Claim3) = H(S(Claim1) + Claim3)$
$Output = S(Claim2) + S(Claim3)$

Figure 10: Simple Dependency Example



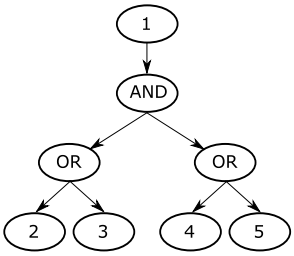
$Claim1 \rightarrow Claim2$ or $Claim3$
$Claim2 \rightarrow Claim4$ or $Claim5$
$Claim3 \rightarrow Claim4$ or $Claim6$
$S(Claim1) = H(Claim1)$
$S(Claim2) = H(S(Claim1) + Claim2)$
$S(Claim3) = H(S(Claim1) + Claim3)$
$S(Claim4) = H(S(Claim2) + S(Claim3) + Claim4)$
$S(Claim5) = H(S(Claim2) + Claim5)$
$S(Claim6) = H(S(Claim3) + Claim6)$
$Output = S(Claim4) + S(Claim5) + S(Claim6)$

Figure 11: Example Showing Multiple Parents

including an AND operation is shown in Figure 12. The AND node has two branches for its two children. A different string for the AND is given to each branch, represented by AND1_1 and AND1_2. (The two AND pieces are shown without the S(x) notation, because they aren't actual vertex nodes in the graph.) When the two branches are XORed together, the result is the actual value of the AND node. For an n-input AND, this is done by generating n-1 random values (each the size of the output of the hash function) and using them as the values for the first n-1 branches. The final branch is the XOR of the rest of the branches and the AND node's value. All of the randomly generated values are included in the string that is hashed to get the AND node's value (to prevent any linear combination attacks against the XOR combination). This is a simple (n, n) secret sharing scheme, which requires the strings of all of the AND node's children to be known to reconstruct the value of the AND node itself. The example also shows how the OR nodes disappear, because their value is equal to their parents' value. This is still true (if slightly more complex) when such an OR node has multiple parents; the parent values are simply concatenated, in the same way multiple parents of any other node are handled.

As an example of requiring combined AND and OR dependencies, imagine a table of claims, with each column containing a different type of claim. Consider the rule that to access an element of the first column requires also showing (at least) one element of every other column. Using our method, this requires a graph containing one node for every element in the table, a single AND node, and one OR node for each column but the first two.

Certain dependencies are not handled by our current approach, including cyclic dependencies, negative dependencies, and operations that cannot be represented as a combination of ANDs and ORs. We do not believe that negative dependencies between released claims is meaningful, since the prover could always perform multiple, independent showings of the signed documents. Cycles are prohibited because it is



$Claim1 \rightarrow (Claim2$ or $Claim3)$ and
$(Claim4$ or $Claim5)$
Is transformed into:
$Claim1 \rightarrow AND1$
$AND1 \rightarrow OR1$ and $OR2$
$OR1 \rightarrow Claim2$ or $Claim3$
$OR2 \rightarrow Claim4$ or $Claim5$
$S(Claim1) = H(Claim1)$
$S(AND1) = H(S(Claim1) + AND1\_1)$
$AND1\_1 =$ Random value, of size $|H|$
$AND1\_2 = S(AND1)$ xor $AND1\_1$
$S(OR1) = AND1\_1$
$S(OR2) = AND1\_2$
$S(Claim2) = H(S(OR1) + Claim2)$
$= H(AND1\_1 + Claim2)$
$S(Claim3) = H(S(OR1) + Claim3)$
$= H(AND1\_1 + Claim3)$
$S(Claim4) = H(S(OR2) + Claim4)$
$= H(AND1\_2 + Claim4)$
$S(Claim5) = H(S(OR2) + Claim5)$
$= H(AND1\_2 + Claim5)$
$Output = S(Claim2) + S(Claim3)+$
$S(Claim4) + S(Claim5)$

Figure 12: Combining AND and OR

Prover provides:
$Output = S(Claim2) + S(Claim3) +$
$S(Claim4) + S(Claim5)$
$S(Claim2) = H(AND1\_1 + Claim2)$
$S(Claim4) = H(AND1\_2 + Claim4)$
$S(AND1) = H(S(Claim1) + AND1\_1)$
$S(Claim1) = H(Claim1)$

Verifier checks:
Output is signed (in the hash tree)
All hash values are correct
$S(AND1) = AND1\_1$ xor $AND1\_2)$

Figure 13: Showing claims 1, 2, and 4

impossible to calculate the values for the nodes in the cycle without breaking the hash function used. [1]

Our implementation detects cycles using Tarjan's algorithm [60] and combines all nodes in each strongly connected component (i.e., all nodes involved in a cycle) into a single node (per component).

4.2.2 Protocols for Usage

In general, a set of claims will have some claims with no dependencies and other groups of claims that are inter-dependent. To handle this situation, we combine the structures described in the previous subsection with the hash-tree-based redactable signature scheme from our prior work. Each group of inter-dependent claims is represented by a DAG and a single signed output value is generated for each such group. Each of these signed output values then becomes a node in the overall hash tree, along with each of the claims that have no dependencies associated with them. As in the prior approach, the certifier signs the root value of this hash tree and places it in

[1] Alternatively, all nodes of a cycle can be thought of as being pieces of a single, larger claim. It is easy to show that this is not true in general, because sets of claims can be separated out, which don't depend on all values in the cycle. On the other hand, a general rule for breaking cycles requires enumerating possible paths as separate nodes, which is unacceptable given the exponential number of possible paths.

a PKI certificate.

To show a claim that has dependencies requires showing more claims to fulfill those dependencies. We refer to a claim and one set of additional claims that fulfills the dependencies as a chain. The term "chain" is not strictly accurate, since the chain will have multiple branches if it has any AND nodes, and those multiple branches may even connect together (ie, not a tree). There can be no loops, per the constraint that dependencies must be in the form of a DAG.

As an example, consider the graph of Figure 12 and the case of showing "Claim1", "Claim2", and "Claim4". The prover must provide to the verifier the input strings that were hashed to create the string values of each of the claims being shown and the AND node on the path, along with the (signed) output value. The input string for a claim node includes the actual data of that claim, so the data for the three claims is included in what is shown. Figure 13 summarizes what the prover shows and what the verifier needs to verify. The only additional values given to the verifier that are not used are $S(Claim3)$ and $S(Claim5)$. These are the string values for the other two leaf nodes, and contain just the hash output. Under the assumption that the hash function is secure (can't be inverted and doesn't leak data), then no data is leaked by these extra strings, except for the knowledge of their existence.

4.2.3 Relation to prior work

The mechanism for handling dependencies obviously shows a family relationship to the Merkle hash tree. It in fact looks like a backward tree, where instead of a root node verifying the values of many leaf nodes, a set of leaf nodes can verify the value of a set of root nodes (and all of the intermediate nodes as well). In most of our examples, the set of leaf nodes is much larger than the set of root nodes, but that is simply selection bias of the examples.

4.3 Security

4.3.1 Threat model

The primary threat our system is designed to resist is the prover and verifier collaborating to cheat the certifier, by violating the dependencies on showing claims. In this case, security is done on a "can prove" basis, where it doesn't matter how the certified data is proven. In particular, the verifier does not have to follow established protocols or intentions. (Put simply, we do not assume that the verifier is an honest player in the system.) We refer to the verifier as suspicious (of the prover), but rule-breaking.

Formal security proofs for the dependency violation attack as well as for forgery attacks and privacy violation attacks are contained in Appendix A. Only informal security arguments are given here, for ease of readability.

4.3.2 Additional details

A secure scheme requires some additional details beyond those described so far. First and foremost, the claims must have random padding to prevent a dictionary attack, because the hash value of unreleased claims is necessarily provided to a verifier. Second, and related, is that nodes should be unambiguous about what they contain. Throughout this paper, the value of nodes is represented by a simple concatenation of values. In our actual implementation, we include additional meta-data identifying the node type and the length of fields. Additional meta-data might also be required in specific applications of the approach. For example, if applied to documents, the order of the claims (words) is important and this information can be provided in the meta-data in the form of sequence numbers. If knowing the sizes of the gaps in the provided text is important, contiguous sequence numbers are used. Otherwise, random increments between consecutive sequence numbers can avoid release of extra information such as the precise number of words that were redacted from a particular section.

4.3.3 Forgery

Forgery covers all cases where the verifier is convinced that a claim is certified by a particular entity, when it was not. Forgery covers several different problems, depending on what part of the system is attacked. For example, a forger can try to attack the hash function to create a bad final or intermediate value. A secure hash function will prevent this type of attack.

A forger can try to pass off data as if it were part of the structure, or vice versa. This style of attack is only possible if the construction of nodes' strings is done in a simplistic manner, such as merely concatenating the values. Because of the additional meta-data, discussed above, this attack is not possible.

A forger could also try to fake a valid looking AND node. The AND node construct of XORed masks is a trivial secret sharing scheme, while having the masks inside of the hashed string is a constraint on the values of shares that are accepted. This constraint is necessary to eliminate the possibility of generalized birthday attacks [64]. The problem of faking an AND node can be reduced to the problem of creating a collision in a hash function defined as $H(x + \{y_i\}) = H'(x + \{y_i\})$ xor $\{y_i\}$, where $H'(x)$ is a hash function (assumed to be secure) and $\{y_i\}$ is an arbitrarily sized set of values. This construct is easily shown to be secure when $H'(x)$ is modeled as a random oracle, and we believe it to be secure for practical instantiations of $H'(x)$

4.3.4 Loss of Privacy

A loss of privacy occurs when releasing some claims or data exposes additional claims or data that the holder did not intend to release. In our system, the most obvious way for additional data to leak is through hashes of unreleased data being (necessarily) revealed. Under the assumption of a secure hash function, no information about the data should be directly leaked by its hash value. However, if the data itself is in a guessable form and of low entropy, then a dictionary attack may be performed against

the hash. Simply adding random padding fixes the problem. The padding can either be independently generated and stored for each data value, or it can be generated using a pseudorandom function along with a random seed value.

4.3.5 Violation of Dependencies

A violation of dependencies occurs when a prover is able to release certified data to a verifier in a way that the verifier can determine that it was certified, while not releasing other data as required by the certifier.

Dependencies are enforced by providing chains, such that each link in the chain must be fully verifiable in order for the next link to be verifiable. Additionally, to verify a (data) link requires the actual data for that node to be known and used by the verifier. These chains overlap, creating the DAG. There are two general ways to violate the dependencies: forging a link in the chain and finding a different way (than following the chain) to prove a claim. The first of those is covered by the subsection on forgery, discussed previously. The second is beyond the scope of this paper, as it covers application-specific side-channels, and is dependent upon the implementation and use of the system.

4.3.6 Other

There is an additional class of attacks that are of less concern, namely attacks by the certifier of the data. The only meaningful attack we can see is the certifier putting a hidden channel into the signature, without the knowledge of the user. As we generally expect the certifier to provide the data, define the dependencies, and create the actual structures of the signature, we consider attempting to identity and prevent all possible hidden channels as beyond the scope of this paper. Since in all applications we can imagine for the approach, the user has an inherent trust relationship with the certifier, we do not consider this issue a serious one.

4.4 Performance Results

Good performance for large systems of dependencies was a primary goal of this work. Two evaluations of the scheme in meeting that goal are provided. The first evaluation is in the form of analytical bounds, while the second is based on experimental results from an implementation of the scheme.

4.4.1 Analytical Bounds

Our scheme provides clear bounds on space and time complexity. There is exactly one vertex node for each claim, and the actual data of each claim is stored and hashed only once. There is a maximum of one vertex for each AND and OR as well (multiple ANDs and ORs may be combined).

For comparison, consider a brute-force approach to handling dependencies. All possible combinations of claims necessary to satisfy a particular claim's release policy can be combined and treated as a single claim. For small graphs, this can actually be quite efficient. However, consider the example given before of a table of claims, with the rule that to access an element of the first column requires also showing (at least) one element of every other column. As noted before, the number of nodes in the graph of this example is equal to the number of items in the table plus one (ignoring the removed OR nodes). Assuming a symmetric table (all columns have the same number of rows), the approximate total size of data being hashed in the creation of the graph is given in Equation 3, where n is the number of rows, k is the number of columns, $|H|$ is the size of a hash:

$$(nk + n + k - 3) \cdot |H| + |\text{all data items}| \tag{3}$$

Therefore, our solution is $\mathbf{O}(nk)$ in space and time, both for dependency graph construction and for claim verification. For comparison, enumerating all possible claim combinations is $\mathbf{O}(n^k)$ in this example.

Later, we discuss an input graph based on dependencies between software packages available in the Ubuntu software repositories. Tests are done on a graph of approximately 45,000 nodes (holding about 25,000 claims). There are more than six hundred million possible path combinations of released claims that would have to be separately calculated and stored under the brute-force approach.

4.4.2 Trade-offs

There are a number of trade-offs that can be made between storage space and speed. For example, separate, smaller DAGs can be calculated and stored for every claim that is involved in a dependency calculation, allowing for quick retrieval and verification of single claims and verification chains. Alternately, a single large DAG (or a small set of medium sized DAGs) can be used, which can greatly increase the amount of data necessary to verify a single claim (and chain), but which is much more efficient to store and verify large sets of claims (with overlapping dependency chains).

The discussion so far has assumed a single, monolithic DAG is used. A single monolithic graph is easier to describe, program, and analyze. However, a multiple graph approach can produce better results in a "best-case" scenario. As such, both a monolithic version and a multiple graph version were implemented for experimental performance testing. To provide the best contrast, the multiple graph version makes and stores a separate DAG for every claim.

4.4.3 Experimental Setup

A Java implementation of the dependency handling portion of the system was evaluated. Tests were performed on a Dell PowerEdge 2900 server with dual Xeon 5150 CPUs running at 2.66 GHz using Sun's 64-bit server JVM, version 1.6.0. The code was single threaded and the server otherwise idle. The minimum time seen over multiple rounds was recorded, to avoid artifacts from just-in-time compilation and garbage collection. SHA-256 was used for the hash function [42].

Two styles of artificial input dependency graphs were used, both rectangular tables. Using this format, it was easy to vary the number of rows, number of columns, and size of data in each claim (constant across all claims in a table to make the results more consistent). The first style matched the example analyzed earlier, where a table of claims has the requirement that to release any claim in the first column requires releasing (at least) one value from each of the other columns as well. This graph is fast to process, as would be expected from the earlier bounds discussion and some simple analysis (most of the nodes in the graph are of low degree, with the connections concentrated in the AND and OR nodes).

The second graph is a table where each column is dependent upon the next column, i.e. showing a claim in any column except the final column requires showing at least one claim from the next column. (For claims in the first column of the table, the behavior of this graph style is the same as for the previous graph style.) This graph is denser in connections than the first graph, with all nodes being of degree n or $2n$.

One large, real-world dataset was tested. This dataset was derived from the dependencies between software packages in the standard Ubuntu Linux repositories. The "Depends" and "Pre-Depends" values were used to determine the dependency graph; other package links ("Recommends", "Suggests", "Replaces", etc) and version information were ignored. A total of 25157 packages were used, representing 21GB of data. Fake empty files of the correct size for each package were created, stored as sparse files on an ext3 formated partition. (The file system reports only 820KB of actual disk space used.) The dependency graph contains 566 cycles containing 2977 nodes, which are condensed down to 566 new nodes. Timings were taken using both short strings, consisting of the package name plus 18 fixed characters, and the full length fake package files.

Table 2: Timings for first graph style (in μs)

Input Table Size			Monolithic Graph		Multiple Graph	
Rows	**Columns**	**Data size**	**Verify chain**	**Verify all**	**Verify chain**	**Verify all**
Small inputs						
4	4	10	360	330	120	450
4	8	10	520	460	200	660
4	16	10	960	890	400	1500
4	32	10	1900	1800	950	3600
Medium inputs						
64	16	100	1700	8200	1300	77,000
64	32	100	3400	17,000	4400	280,000
64	64	100	6800	34,000	19,000	1,200,000
64	128	100	15,000	74,000	77,000	5,000,000

4.4.4 Experimental Results

4.4.4.1 Absolute timings

Handling of dependencies is efficient for most cases, as can be seen in Table 4.4.4.1, which shows timing data for the first graph style. Most of the operations take at most 10 milliseconds. However, the multiple graph approach can be inefficient when the number of claims to be verified is very large. For example, for a 64×128 table with 100 bytes per entry, the multiple graph approach requires 5 seconds to verify *all* claims, while the monolithic graph approach only takes 74 milliseconds for the same case.

The second graph style has the same complexity bound as the first style, but is much slower in practice due to the high density of connections. When analyzing this style, we found that the time to pre-generate all of the chains in the multiple graph approach makes it impractical. Thus, we report only the monolithic graph results for this case. Figure 14 shows the time (in milliseconds) for verifying a single chain in a monolithic graph (compare to the first timing column in Table 4.4.4.1) using the second graph style. The time is very small (at most tens of milliseconds), even when the numbers of rows and columns become large. Figure 15 shows the time (in

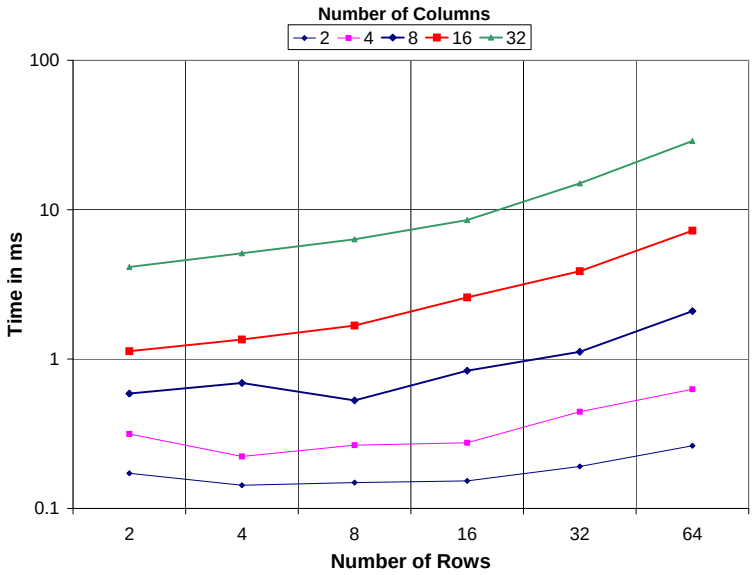


Figure 14: Verifying a Chain in 2nd Graph Style

milliseconds) for verifying the entire monolithic graph (compare to the second column in Table 4.4.4.1) using the same data. This time is also reasonable, being less than one second even for large cases.

The Ubunutu package dependency graph was only timed using the monolithic graph approach. The timings are shown in Table 4.4.4.1. The difference between the short string and full data versions is just the amount of data in each node that has to be hashed. As can be seen from comparing the two lines in the table, the full data version takes approximately an order of magnitude longer than the short strings version. The additional time is primarily hashing the data. Thus, for the full data version, the overhead of graph manipulation is small compared to the time needed to simply hash the data.

Unlike the two table-based graphs, the length of chains in the Ubunutu package graph varies considerably. The measurements were taken by selecting 2000 pseudo-random chains out of the full graph. The same pseudorandom paths were used in both the short string and full data cases. The average path length was 17 nodes, while the median was 8 nodes.

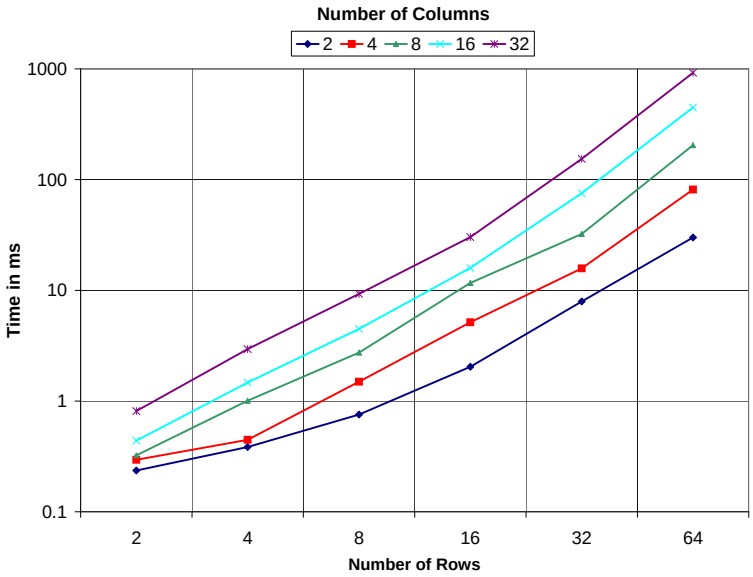


Figure 15: Verifying Full Graph in 2nd Graph Style

Table 3: Timings for the Ubuntu Package Graph (in μs)

Input Type	Produce Graph	Verify Graph	Produce Chain		Verify Chain	
			Average	Median	Average	Median
Short strings	13,000,000	16,000,000	178	81	24,000	17,000
Full data	280,000,000	280,000,000	209	83	240,000	180,000

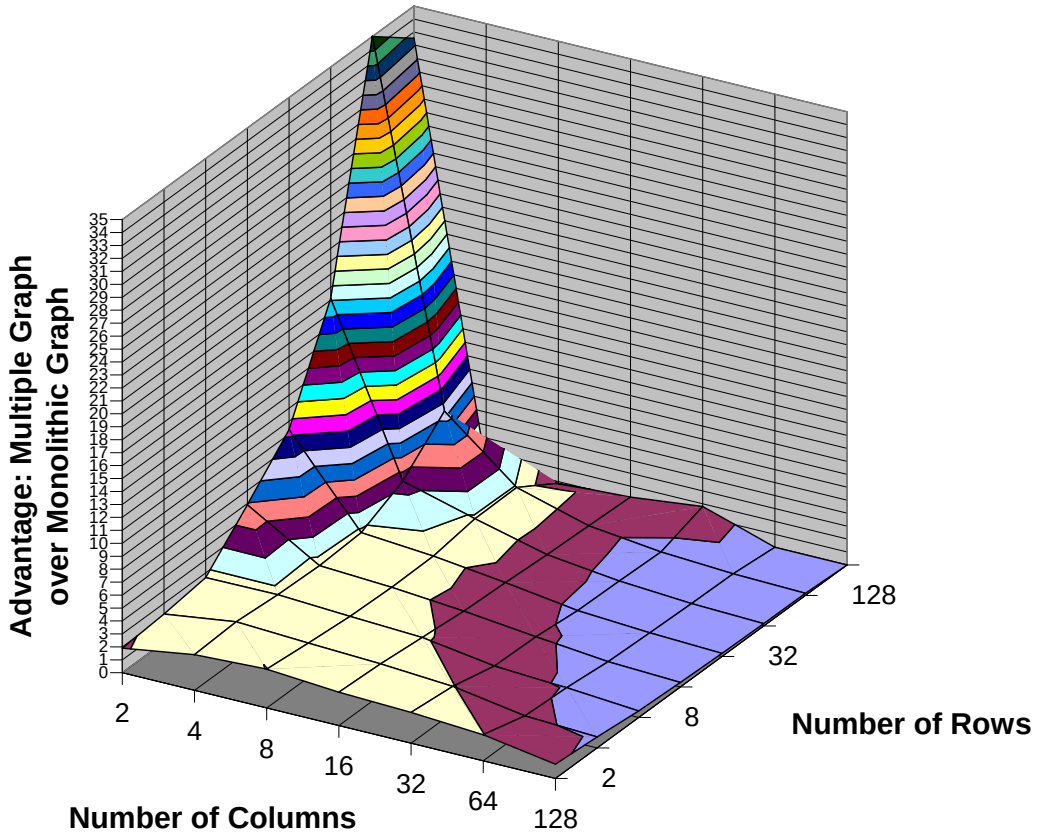


Figure 16: Relative Time to Verify a Single Dependency Chain

4.4.4.2 Relative timings

Here, we consider the relative performances of the monolithic and multiple graph approaches. Figure 16 shows the speedup of the multiple graph approach over the monolithic approach. (This graph is based on the first graph style and a data size of 10 bytes.) The multiple graph approach provides a significant advantage—about 35 times faster for the peak in the figure—for some parameter values. In particular, it does best with a large number of rows and a small number of columns. The area to the right of the first contour line is where the monolithic graph performs better. Thus, the optimum implementation approach is dependent on the nature of the dependencies. However, from a worst-case standpoint, the monolithic graph implementation is superior.

4.5 Preserving Order and Gap Information

In some applications using a redactable signature, the order of the claims is important, for example when each claim is a word or sentence in a single document. Additionally, the application may require that a verifier can determine when two apparently adjacent claims originally had one or more claims between them. Supporting such applications using our scheme is easily accomplished via meta-data added to the input string of each data node. In the case where both order and gap information must be available, the nodes are simply numbered in order. In the case where order must be preserved, but not gap information, a random increment can be used. In either case, the meta-data is hashed along with the rest of the input string to give the string value for the node.

CHAPTER V

CREDENTIAL-HOLDING IDENTITY AGENTS

The previous two chapters describe ways to efficiently store and assert a large number of claims. While this provides many benefits, it also has at least one major drawback, especially in the case where the claims are part of a credential. An attacker needs only to compromise a single credential to gain control over all of the claims in it. Additionally, there is no mechanism specified so far for detecting or dealing with a credential being compromised. This chapter describes one solution to these problems, called GUIDE-ME.

5.1 Introduction

The Georgia Tech User-centric IDEntity Management Environment (GUIDE-ME) is a system for storing and using long-lived credentials with relying parties that a user may never have communicated with before. As part of that purpose, it provides:

- credentials that are efficiently verifiable, copy resistant, and allow the user to dynamically select which personal attributes to disclose in a given transaction
- a mechanism for users (or their agents) to monitor transactions involving their identities, for the purpose of detecting unauthorized use
- a mechanism for users to set policies on when their information can be used
- a remote identity agent than implements some of the above features, offloads identity related tasks, and is always on-line to respond to identity related tasks

5.2 Overview

The basic idea of the GUIDE-ME system is to split a user's credentials, storing the bulk of the credentials on a network-attached server, while keeping the user's private key under the user's more direct control. Using the minimal disclosure credentials from Chapter 3, it is easy to store the claims and hash tree structure on the network-attached server, while keeping the private key on a local device, physically in the user's possession. Additionally, by having such credentials available, GUIDE-ME is able to define protocols that can protect against man-in-the-middle attacks better than the current de facto standards of SSH [69] and TLS/SSL [20] when communicating with a not-seen-before relying party.

5.3 Architecture

The GUIDE-ME architecture consists of three entities: a local identity agent, a remote credential-holding identity agent, and a relying party. The three entities and the communication between them is shown in Figure 18. All parts of the prototype system are written in Java, although the local identity agent is encapsulated in an executable file (.exe) when used on the Microsoft Windows platform for ease of invocation.

5.3.1 Local Identity Agent

The local identity agent is a software program presumed to be running on a device in the user's physical possession. The device may be a smart phone, PDA, specialized hardware token, or computer. Because the local agent may be running on a resource constrained device, the protocols (described later) were designed to minimize the load on it. Because the local agent is likely to be running on a device that is easily lost, stolen, or compromised, it does not hold the actual credentials, but only indexes of the user's credentials and claims. It also holds the user's private key, to keep the user's private key as close to them as possible.

The local identity agent is the primary software program the user interacts with in the GUIDE-ME setting. In the prototype, the local agent is invoked by the user's webbrowser when it encounters a pre-determined file type (a .idm file). The file tells the local agent the identify of the relying party (both a certificate and a URL), the attributes being requested, and the session ID (as seen by the relying party). The local agent compares the request to its indexes of what credentials the user has and checks its policy for whether it has sent information to this specific relying party before. If the user's policy requires a password for any of the claims before released, the agent will query the user for the password, as shown in Figure 17. Passwords given to the local agent are never stored; they are only sent to the remote agent for verification there. The local agent will also (separately) request confirmation for sending any attributes which have not be sent to the specific relying party previously.

5.3.2 Remote, Credential-Holding Identity Agent

The remote, credential-holding identity agent is a software program presumed to be running on a device in the network cloud. In GUIDE-ME, it holds the bulk of credentials for the user, but doesn't hold keys necessary for using those credentials. Under variants of the architecture, the agent can hold encrypted credentials, credential shares, and/or key shares, as discussed in Section 7.3.3.

The remote agent has its own public/private keypair, so that it can sign messages, without having access to the user's private key. The remote agent's public key is given by the user to the relying party, as per the message descriptions in Section 5.4.1. The remote agent's key does not need to be signed by a trusted authority, since it is simply a delegate of the user.

The remote agent is considered to be "semi-trusted". It has the ability to break the user's privacy (through the contents of the credentials it holds), but it doesn't have the ability to impersonate the user. Since most users are unlikely to have the

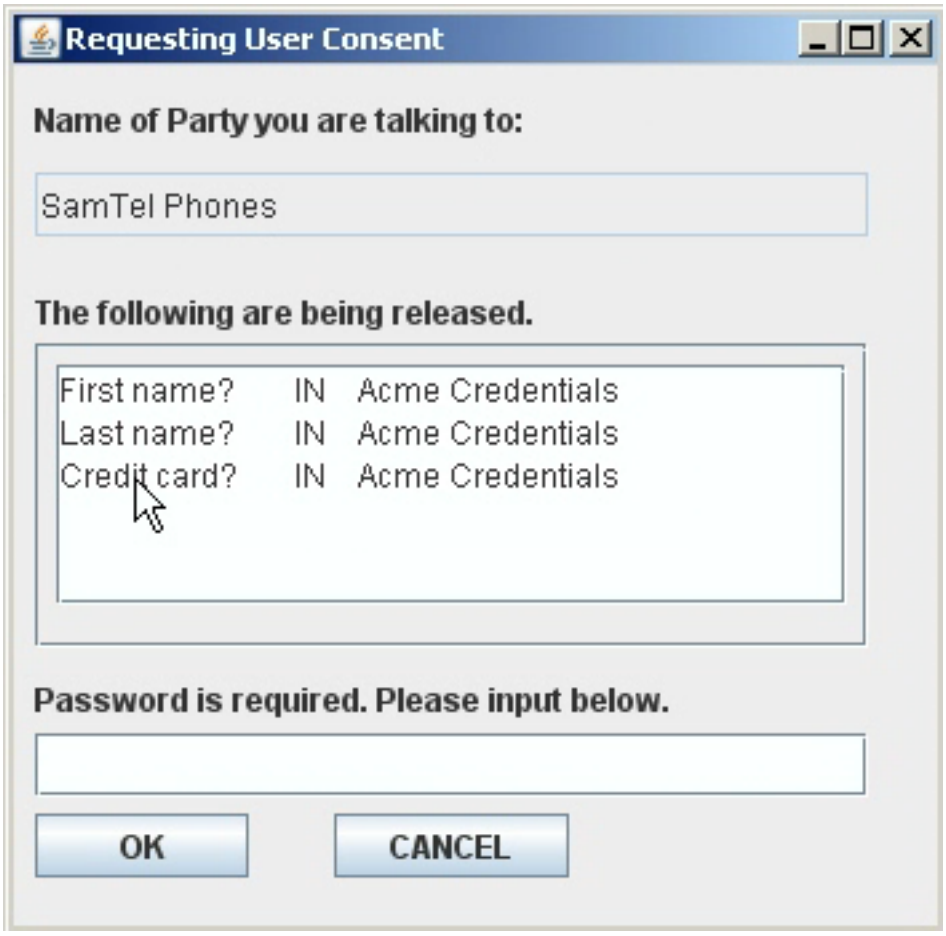


Figure 17: GUIDE-ME dialog box requesting a password.

ability to run and maintain their own network-resident server, the remote agent will likely be run by a third party, such as an employer, bank, internet service provider, or dedicated agent host.

GUIDE-ME's remote agent implementation is built around a webserver, using Sun's HttpServer class, and a database, using the Berkeley DB engine. The database stores the credentials, along with meta-data about their contents, and optional passwords for additional security, all tied to specific user names.

Because the remote agent is involved in transactions, it is in a position to monitor those transactions for usual activity. Unfortunately, there are a lot of caveats associated with this ability. First, once an attacker has gotten part of a credential and controls the user's private key, they can re-use the known part of the credential without involving the remote agent. So the remote agent might only have a single transaction to identify as suspicious. Second, users may make transactions that they do not want records of. Both of these are addressed by a modified GUIDE-ME architecture in [36].

5.3.3 Relying Party

The relying party is the entity that receives the credentials from the user. As stated in the Introduction, a replying party can be a peer in peer-to-peer communication, but the GUIDE-ME implementation is aimed at client-server settings. The relying party is assumed to be an on-line service provider of some type. (The implementation includes a fake e-commerce website, to use in demonstrations.)

GUIDE-ME's replying party implementation is built around a webserver using Sun's HttpServer class. Authentication requirements are specified using a simple plain-text file, which lists paths and the attributes required to access them. Attributes which have been proven (by providing a credential with matching claims) are stored, connected to the user's session. Cookies are used to maintain the session information.

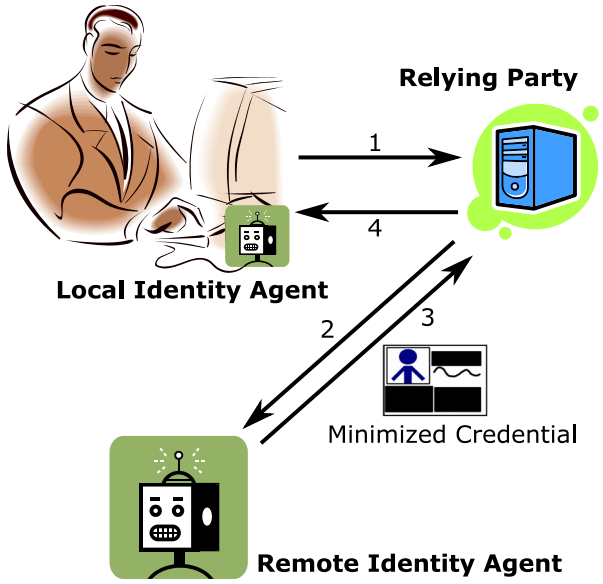


Figure 18: GUIDE-ME Entity and Communication Architecture

(No sensitive information is stored in the cookies; only randomly generated identifiers. If a client presents with an identifier that is not known, a new one is generated.)

5.4 Protocols

Figure 18 shows the communication that takes place, as seen by the encapsulating protocol (HTTP in the implementation). An initial two-way communication takes place between the local agent and the relying party, in which the parameters for the authentication are determined. This initial communication is not labeled in the figure, because the form and number of steps can vary widely. In particular, the local agent and relying party may have a many-step negotiation over what credentials must be shown.

5.4.1 Messages

As seen by the entities, there is an additional message to the ones shown in Figure 18, going from the local agent to the remote agent. This message, referred to as an authorization token, is included as a part of messages 1 and 2 in the figure. The authorization token is required to be encrypted (using the remote agent's public

key). The other messages are optionally encrypted; requiring them to be encrypted would be inefficient, since the entire protocol may be encapsulated in a secure tunnel such as TLS/SSL [20]. [1]

All messages are signed by the sender and contain a session ID, which is a simple concatenation of user-generated and relying party-generated nonces. The messages are:

1. Local agent to relying party, contains the public keys of all three entities (local agent, remote agent, and relying party), the authorization token for the remote agent, and a contact address for the remote agent.

2. Relying party to remote agent, contains the authorization token for the remote agent.

3. Remote agent to relying party, contains the credentials (or a message explaining why the authorization was rejected).

4. Relying party to local agent, contains a notification of success or failure.

- Local agent to remote agent, contains user and device IDs, the relying party's public key, information about which credentials to send, and optional additional authorization passwords.

For simplicity, the above protocol is considered the canonical protocol for GUIDE-ME. However, the implementation allows for the local agent to directly contact the remote agent and get its response with the credentials. The relying party will accept the credentials directly from the local agent.

[1] Note that while encryption is not required for the security of the protocols, it is required for maintaining the privacy of the content of the sent credentials. Privacy using an external tunnel requires connecting at least one public key from the connection to the tunnel. While this is easy in theory, in practice, Microsoft Cardspace was forced to make that feature optional, because many relying parties couldn't support it. [6]

The messages are implemented as specialized classes, extending a base AuthorizationToken class and encapsulated in an EncryptedToken object.

5.4.2 Security

The security theme of the GUIDE-ME protocols is to cryptographically connect the entities, the credentials, and the specific session. These connections can entirely prevent man-in-the-middle attacks. An outline of a proof of security for the GUIDE-ME protocols can be found in Appendix B.

Note that unless the carrier protocol between the local agent and relying party is encrypted and/or authenticated (i.e., every message is signed) a man-in-the-middle can trivially manipulate the session after the GUIDE-ME specific protocol has completed. This problem is outside the scope of the general GUIDE-ME protocols, but is trivially solved for specific uses of it. A specific and complete protocol needs only to specify cryptographic signing of all messages [2], with the signing key(s) tied to the GUIDE-ME protocol.

[2]Messages should be signed using a message authentication code (MAC) and a shared key, and not directly with an entity's private key, for efficiency.

CHAPTER VI

GENERALIZED, USER-CENTRIC AGENTS AND APPLICATIONS TO PERSONAL HEALTH RECORDS

6.1 Motivation

In working on the GUIDE-ME architecture, there were a variety of options, extensions, and applications discussed. In order to best allow for various options and extensions, the underlying architecture needed to be more general and provide an application programming interface (API).

6.2 GUCIdA

A Generalized, User-Centric Identity Agent (GUCIdA) framework was developed. The main areas of functionality provided by the framework are communications, authentication, credential handling, and a database interface. GUCIdA is written entirely in Java, re-using a lot of code from GUIDE-ME. Extensions are done as a plugin architecture, where plugins are discovered and loaded at run time. Figure 19 shows a high level view of GUCIdA, showing the communications (lines to clients and service providers), plugins (colored blocks), and database.

6.2.1 Communications

The communications functionality is centered around HTTP, as a way to allow maximum compatibility with existing software and infrastructure. No arbitrary (TCP, UDP, or other) communications API is provided, since Java already provides such API's.

GUCIdA provides a pair of classes (AuthorizationToken and EncryptedToken; inherited from GUIDE-ME) as a standardized way to produce secure messages. All

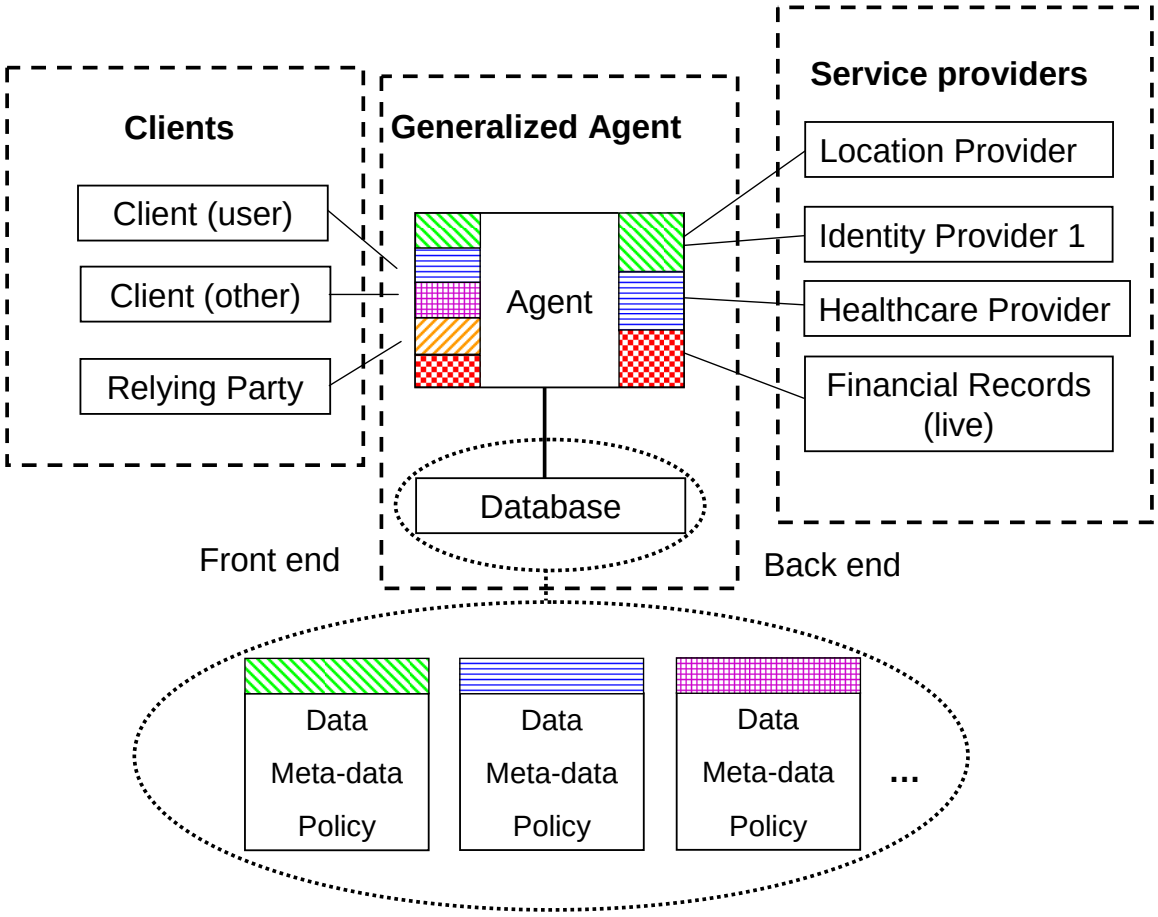


Figure 19: High-Level View of GUCIdA Architecture.

secure messages used in Medvault (described in Section 6.4) make use of these classes.

GUCIdA uses the Sun httpserver class to service incoming communications for all plugins via a single server instance. Path information is used to direct requests to the correct plugin. Convenience methods for both error and normal responses are provided, along with an infrastructure for session handling, redirection, and authentication. Session handling allows plugins to associate arbitrary data with each connection, where a connection is denoted by a stored cookie. The data namespaced, so that different plugins can't interfere with each other's data. Additionally, authentication information is automatically associated with sessions, as discussed below.

GUCIdA uses the Java HttpURLConnection class for outgoing connections, but does not encapsulate this interface in any additional APIs. Plugins are free to follow this usage example and all currently existing ones do.

GUCIdA is primarily single-threaded, due to the httpserver class instance being single threaded. As an exception to this, multiple instances of the httpserver class can be run simultaneously, each in its own thread. Under this method of execution, all GUCIdA-based entities in the Medvault demonstration can be run in a single program instance. Entities are distinguished via port numbers.

6.2.2 Authentication

GUCIdA automatically handles some parts of authentication—providing a mechanism for requesting and receiving credentials and maintaining lists of verified claims for each connection—and provides an API for other parts, most prominently for specifying access control rules.

Plugins request (via a method call) authentication, specified in terms of claims and a specific user session. Plugins also specify a destination URL, for after a user has provided the requested claims. GUCIdA handles generating the actual request, receiving the response, verifying any received credentials, storing tables of all verified

claims (one table per user session), and redirecting the user to the correction destination when authentication is complete. Limiting the acceptance of claims according to what identity provider is validating each claim is not yet implemented.

Two different forms of credentials are handled by the GUCIdA prototype. The two forms are the MHT credentials from Chapter 3 and a simpler signed flat list of claims meant for dynamic attributes. Support for additional forms of credential can easily be added, of course.

Claims from received credentials are stored in a dedicated structure (to allow searching by full or partial matching), tied to a specific session ID, and accessible by all plugins.

An API is also provided for storing, searching, and retrieving credentials from the database. This API both handles common functionality and provides a cross-plugin handling of credentials. For example, the default credential holding plugin described in Section 6.3.1 provides a web interface for users to upload credentials. Other plugins, such as the attribute provider described in Section 6.4.2.4, can then use the credentials, without having to provide their own interface for inserting credentials into the database. (Since most database access is namespaced to prevent cross-plugin interference, this cooperation would otherwise not be possible.)

The policy engine, described in Section 6.2.4 is available for use by plugins, but the only plugin currently using it is the patient's agent, described in Section 6.4.2.3. There is no automatic handling of credentials or access control with the policy engine, because resources can map to other things than just paths.[1]

[1]One goal of the Medvault system is fine grained access control. Using the policy engine allows large sets of resource identifiers to be passed internally. If resources were only identified by paths in URLs, the URLs could easily be longer than allowed by specifications.

6.2.3 Database Interface

GUCIdA provides a simple database interface to plugins, with an emphasis on security. The interface aims for absolute prevention of SQL injection attacks. The first of the primary two mechanism used is validating that table and column names contain only valid characters. (This white-listing approach is in contrast to approaches that escape or filter bad characters.) The second mechanism is to use Java PreparedStatements, so that all search keys are escaped by the database interface. As an additional general measure, namespacing is used such that each plugin has its own view of the database and cannot corrupt tables belonging to other plugins. [2]

In order to better abstract away the database backend, database handles are not returned to plugins. Instead, most operations convert results to arrays of objects that are returned. Specific operations that can be done well out-of-core (like hashing entries in the database) are done out-of-core for memory efficiency.

Due to the prototype nature of the GUCIdA implementation, the database API is not complete, containing only calls which are used by existing plugins (or previous versions of the plugins).

6.2.4 Policy Engine

GUCIdA contains a policy engine, updated from a version in GUIDE-ME, built on top of the Sun XACML engine. The policy engine is used to provide intelligent access control to resources, beyond the simple claims and paths system which GUCIdA also provides. Currently, the policy engine is only used for medical information access control in the Medvault system. (Technical details omitted, because the policy engine is primarily the work of students other than me.)

[2] Other mechanisms are provided when plugins need to share database data, such as for the stored credentials.

6.3 Implemented Plugins

A modest number of plugins for GUCIdA have already been written. Many of them are part of the Medvault system (described in Section 6.4) or ported from another environment, such as GUIDE-ME. The most interesting non-Medvault plugin[3] is the credential holder plugin.

6.3.1 (Remote) Credential-Holding Plugin

The credential holding plugin provides two main functions. First, it provides an interface to manage stored credentials. Via a webpage, users can browse their credentials and upload new or updated credentials. The credentials are stored in GUCIdA's common credential database, where they can be accessed by any plugin. Second, the plugin uses an updated version of the back-end engine of the remote identity agent from GUIDE-ME, so it can act exactly like that agent, except for using GUCIdA'S common credential database API, instead of the dedicated interface and databases used by GUIDE-ME's implementation.

The database schema for storing the credentials consists of two tables. The first table holds the binary data for the credentials, the owner's user name, and the validity period, along with a unique identifier derived from the credential and user name. The second table holds the text of the claims, in both full and shortened form, an optional password (stored on a per-claim basis), and credential and claim unique identifiers.

6.4 Medvault

Medvault is a system and framework for patient-centric health record sharing and access.

The Medvault system is strongly inspired by two uses cases produced by the US Department of Health and Human Services' Office of the National Coordinator

[3]The credential holding plugin is indirectly used in Medvault, as it is the easiest way to add credentials to the database, where they are then used by the Medvault system.

for Health Information Technology. The two cases deal with emergency access to electronic medical records [44] and consumer access to their own medical records [45].

A simplified view of the Medvault system in show in Figure 20. It shows the various entities involved and the communications links between them. The two rectangles highlight trust domains. Communication within a trust domain is assumed to be secured, either because both entities within the domain are running on the same physical hardware, or because communication is encrypted and signed via a pre-established relationship (e.g., known IPSEC or TLS/SSL certificates), and is not explicitly addressed by the Medvault protocols.

An alternate, more complete (but also more idealized) view of the Medvault system is shown in Figure 21. (For size/space reasons, this figure is missing the Health Information Exchange, described in Section 6.4.2.5.) This view of the Medvault system highlights having multiple attribute providers, each a different entity providing different (dynamic or short-lived) credentials. It also shows the policy engine, present in the patient's agent, that determines whether a requester gets access and what may be shown. Lastly, this view highlights the sources of data going into the back end personal health-record repository.

The Medvault prototype implementation is built on top of the GUCIdA framework (and comprises most of the plugins currently written for the framework).

6.4.1 Source verifiability

If a patient is presenting a copy of his/her own medical record to a medical provider in a situation where the medical provider is expected to provide treatment based on the information, it is reasonable to ask how much trust the medical provider should put into such a record. In the most obvious example, a patient may manufacture a fake past medical problem (accident, surgery, back trouble) to get a prescription under false pretenses. However, there are many other circumstances where a patient

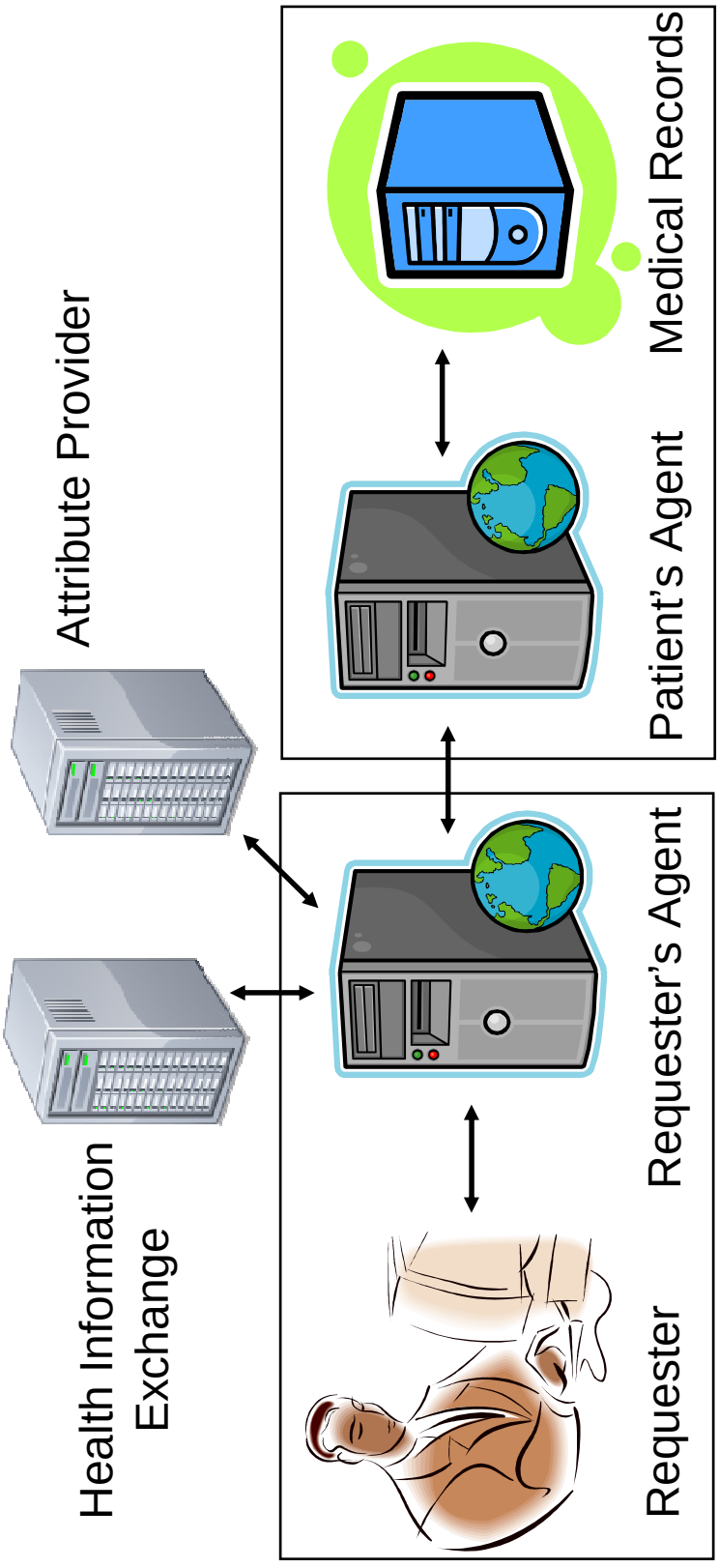


Figure 20: Medvault prototype architecture showing communications.

Figure 21: High-level Medvault Architecture

might be tempted to forge or alter records. For example, a patient who has chosen not to get vaccinations (or not to vaccinate a child) might forge a vaccination record to avoid legal/contractual requirements or arguments with medical providers. A patient receiving psychiatric medication might alter the record to show a condition that they consider less stigmatizing. All of these can have serious consequences for both the patient and the larger community. Medvault addresses this problem through source verifiable records.

In current practice, a source verifiable record usually means one in which it is possible to go back to the originator of the record to check is veracity. This is usually done over insecure channels such as telephone and fax. Besides the low security of the current practice, there is the obvious problem of the source being unavailable due to time constraints (for example, nobody in the office during the weekend or at night), no longer being in existence (for example, a doctor retired or moved from private practice to teaching), or communications difficulties (phone number/address changed; source is not willing to give information over an unsecured channel).

Here, a source verifiable record is one that is cryptographically signed by its originator, such that its authenticity and integrity can be checked without communicating with the source. Along with the cryptographic signature, a chain of trust to well known and long-lived authorities should be stored by the patient. An example of a chain of trust is the certificate path in X.509 [29].

Source verifiability, by itself, is not difficult to achieve. A health care provider can simply sign an entire set of records when providing them to a repository. A third party can then verify authenticity and integrity of the set of records. However, our patient-centric approach dictates that the patient maintain control over which individual records are disclosed within a particular context. Thus, it is necessary to allow disclosure of some subset of these records, while still providing source verifiability. This selective disclosure with verifiability is difficult to achieve in a practical manner.

One trivial approach is to have the health care provider sign each individual item in a health care record when providing it to a repository. However, this approach is not scalable, because a large record could have tens of thousands of individual items in it, and the provider would have to generate that many signatures before sending the record to the repository. Third parties receiving health information would also have to verify many signatures with this approach. An obvious solution is to use the credentials from Chapter 3 and/or Chapter 4 to provide the select disclosure.

In the prototype, Merkle hash-tree credentials can be created across the medical record database. When the records are read, pieces of the hash-tree (called "strands", as described in Section 3.4.1) are read at the same time. The PKI certificates [4] needed for verification are also read. If a record needs to be verified, the strands can be individually verified (as done in the current implementation) or combined together into (one or more) partial trees and verified in bulk. The certificate(s) must also be verified.

6.4.2 Medvault Plugins/Entities

6.4.2.1 Requester

The requester is the person who is trying to access a patient's medical records. We generally assume the requester to be a medical provider, but the system is by no means so limited. Policies can be setup to allow family, friends, and others to access parts of a patient's records.

There is no plugin or software for the requester. Instead, the requester uses a standard, unmodified webbrowser to access the system.

[4] The number of certificates will vary; a single certificate may cover all records, or each record may have its own certificate. The current implementation produces one certificate per table.

6.4.2.2 Requester's Agent

The requester's agent—named "EMT's agent" in the prototype code, because of its usage in the demo scenario for which it was created—acts as a processing and communications agent for an EMT or other medical personnel in the Medvault system. By pushing that functionality to a network-resident agent, the end user (EMT, nurse, or doctor) can use an unmodified webbrowser for interacting with the system. Because the communication is with a fixed agent, standard SSL with a pre-installed certificate is sufficient to secure the connection from the user's device to their remote agent. The communication from the requester's agent to other entities can be secured via pre-known certificates (for the health information exchange and attribute provider) or the credential-session-tied protocols, inherited from GUIDE-ME and described in Section 5.4.1.

In the Medvault prototype, the requester's agent is the central point for communications and activity, as can be seen from Figure 20. Figure 20 shows a slightly simplified view of the Medvault architecture, not including the location service. The two rectangles highlight trust domains.

Once a user/requester has logged into their agent, it monitors connections through it, handling authentication automatically (when possible). The prototype implementation does this via proxying the actual HTTP sessions, modifying headers as necessary, and actively interrupting the communications flow only when it recognizes a special message such as a communications request or signed medical record.

The requester's agent generates most of the user interface, consisting of webpages, that a user/requester sees. This includes translating responses from other entities (including the health information exchange, the patient's agent, and the back-end EHR repository) into human-readable HTML when necessary. For signed (source-verifiable) medical records, the agent will also verify the signatures and color the returned records to indicate that they have been verified (with different colors for

good, bad, missing, and unknown signatures). Some of the webpages will be generated directly by other entities and simply passed along (proxied) by the requester's agent.

6.4.2.3 Patient's agent

Medvault separates out a "proxy" to the medical database whose primary purpose is to handle authentication for access to the medical records; this is referred to as the patient's agent, because it should be under the patient's control. In many cases, the authentication system will be built into the database, and no separate proxy/agent will be needed. However, by having a proxy, Medvault allows the backend record storage to be ignorant of the workings of the medvault system. The records can be retrieved from a professional EHR system, from a managed PHR system[5], or a user-run PHR.

The proxy/agent understands the protocols and credentials of the medvault system, holds the policy engine, and monitors access to the patient's records. It is assumed to have a secure connection to the actual record holder (either a local connection, or a pre-secured link.

6.4.2.4 Attribute Provider

An attribute provider is a trusted entity[6] in the Medvault system that provides short-lived credentials to medical personnel based on their long term credentials and other live information, including the location of entities. Examples of credentials provided include medical credentials, employer, real-time locations, and real-time dispatch confirmation. Real-time information on location and other data is an example of the type of safeguards patients may demand for "break the glass" access to their medical records.

[5]Such as Google Health [37] or Microsoft HealthVault[41]

[6]The architecture and prototype are designed to allow for multiple attribute providers, but for simplicity, only one is used in demonstrations.

6.4.2.5 Health Information Exchange

The health information exchange comes from the US DHHS scenarios mentioned previously. There, the health information exchange is defined as, "A multi-stakeholder entity that enables the movement of health-related data within state, regional, or non-jurisdictional participant groups" [45]. In the Medvault prototype, the health information exchange (referred to as the heath information service in the code) provides services to locate patients' records (via their agents, generally) and attribute providers that can certify specific attributes.

In the prototype implementation, each patient has a unique identifier which is used to look them up. In real life usage, an EMT or other emergency personnel is likely to have to use whatever information is at hand (e.g., name and address, driver's license, phone number from the patient's cell phone) to identify the patient. The health information exchange includes a patient locater service to identify the patient based on such incomplete identifiers.

6.4.2.6 Medvault Medical Record Holder

The medical record holder plugin stores the actual medical records used in the Medvault system. The records[7] are stored in the GUCIdA database using a schema inspired (though greatly simplified) from database schema used by the Veterans Affairs' VistA [62] package. Signatures for the data are stored in extra "shadow" tables that mirror the layout of the medical record schema, in order to provide source verifiability, as discussed in Section 6.4.1 and Section 6.4.4.

[7]No real medical records have been used, for privacy reasons. The records used contain both manually created records and pseudo-randomly generated records.

6.4.2.7 Location Service

The location service is a plugin used in the Medvault demo that stands in for a real life location service. The plugin was ported from a webservice[8], and it uses the Google Maps API to show and set the locations of the entities. When the demonstrator sets an entity's location (by moving a marker on the map), the new location is passed to the attribute provider.

The location service is generally run on the same GUCIdA instance/thread as the health information exchange, as a point of convenience. The two services do not communicate directly or interfere.

A screen shot of the location service's interface is shown in Figure 22. The red markers represent medical personnel and are movable. The green marker represents the current patient and is movable. The blue marker represents a hospital and is immobile.

6.4.3 Interface

As discussed above, the requester's agent produces the user interface seen before the requester gets to the patients' records. The start page is shown in Figure 23, where the requester logs in and specifies the patient. (As discussed in Section 6.4.2.5, a unique patient identifier is used in the prototype, instead of having patients searched for by name, address, and other information.) The requester's agent will then retrieve a list of health record repositories for the given patient and prompt the requester to choose one (not shown).

The request for records goes to the appropriate repository (possibly through a patient's agent), and the requester's credentials will be requested and sent.

[8]Written by another student using the GlassFish environment. As a comment on the power of the GUCIdA framework, the porting process was very quick and required few changes to the code.

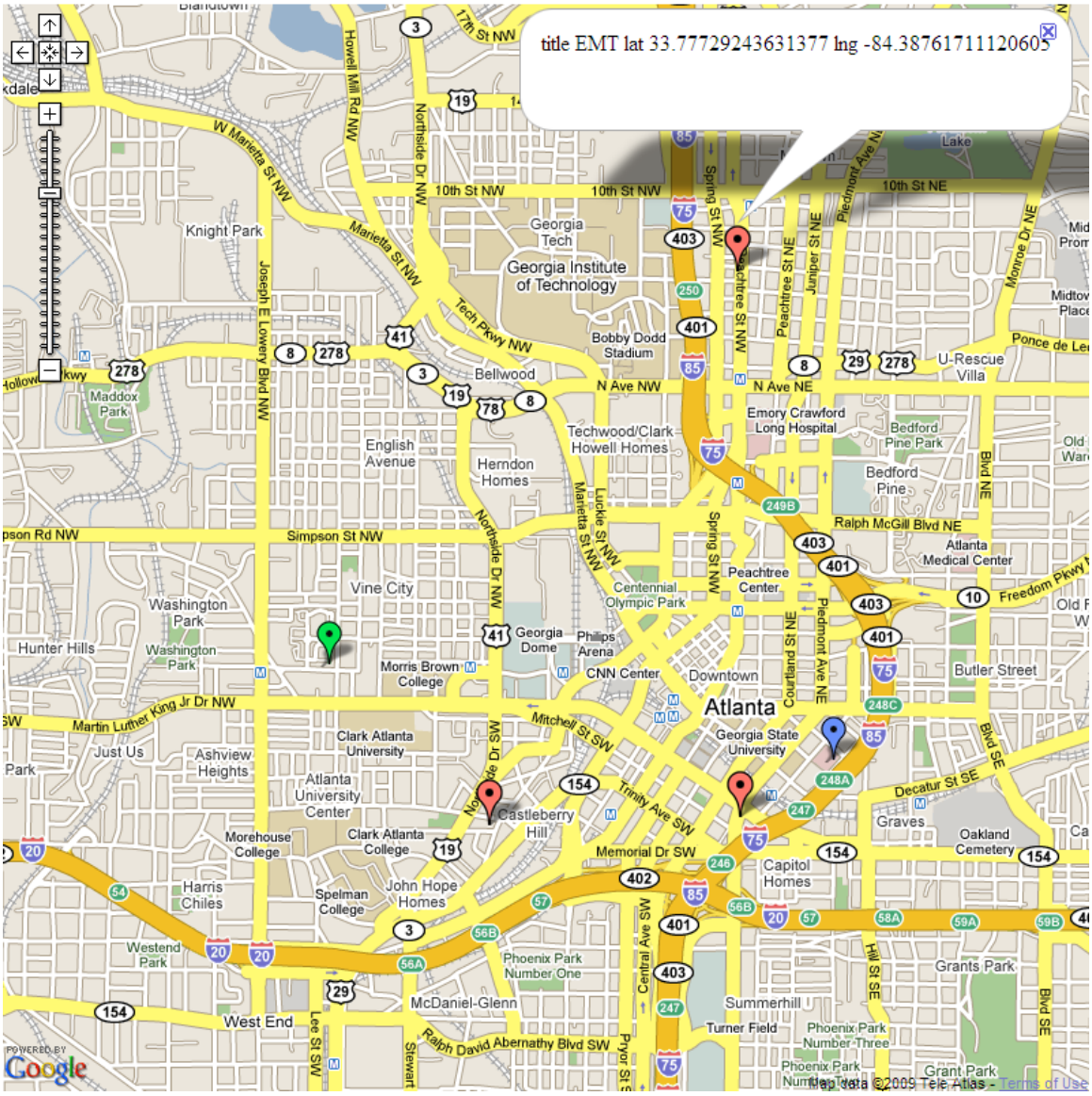


Figure 22: Screen shot of the demonstration location service.

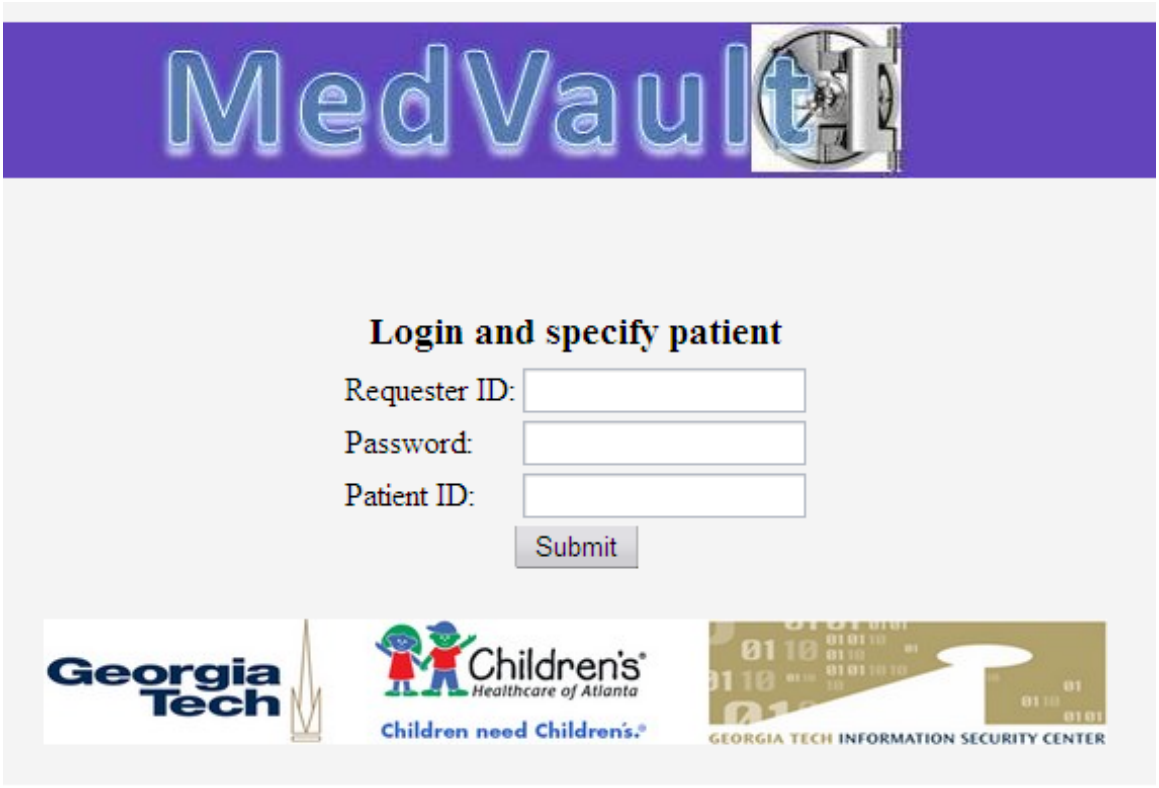


Figure 23: Screen shot of the requester's agent login screen.

Depending on what credentials the requester presents, access may be denied outright, or they may see different views of the patient's records. For example, Figure 24 and Figure 25 show the same patient's records, but under different levels of access. The first requester has presented doctor's credentials and is shown more, while the second requester has presented emergency medical technician credentials and is shown less. These two views are generated by the requester's agent, based on data sent by the backend medical record repository. (Pages without signed records are generated directly by the medical record repository and proxied back to the requester.)

6.4.4 Evaluation of verified database

The source-verifiable database was compared to reading the same database without the signatures, in terms of performance. Timings were done from the start of the method execution until the response was ready to transmit, for reading the records of

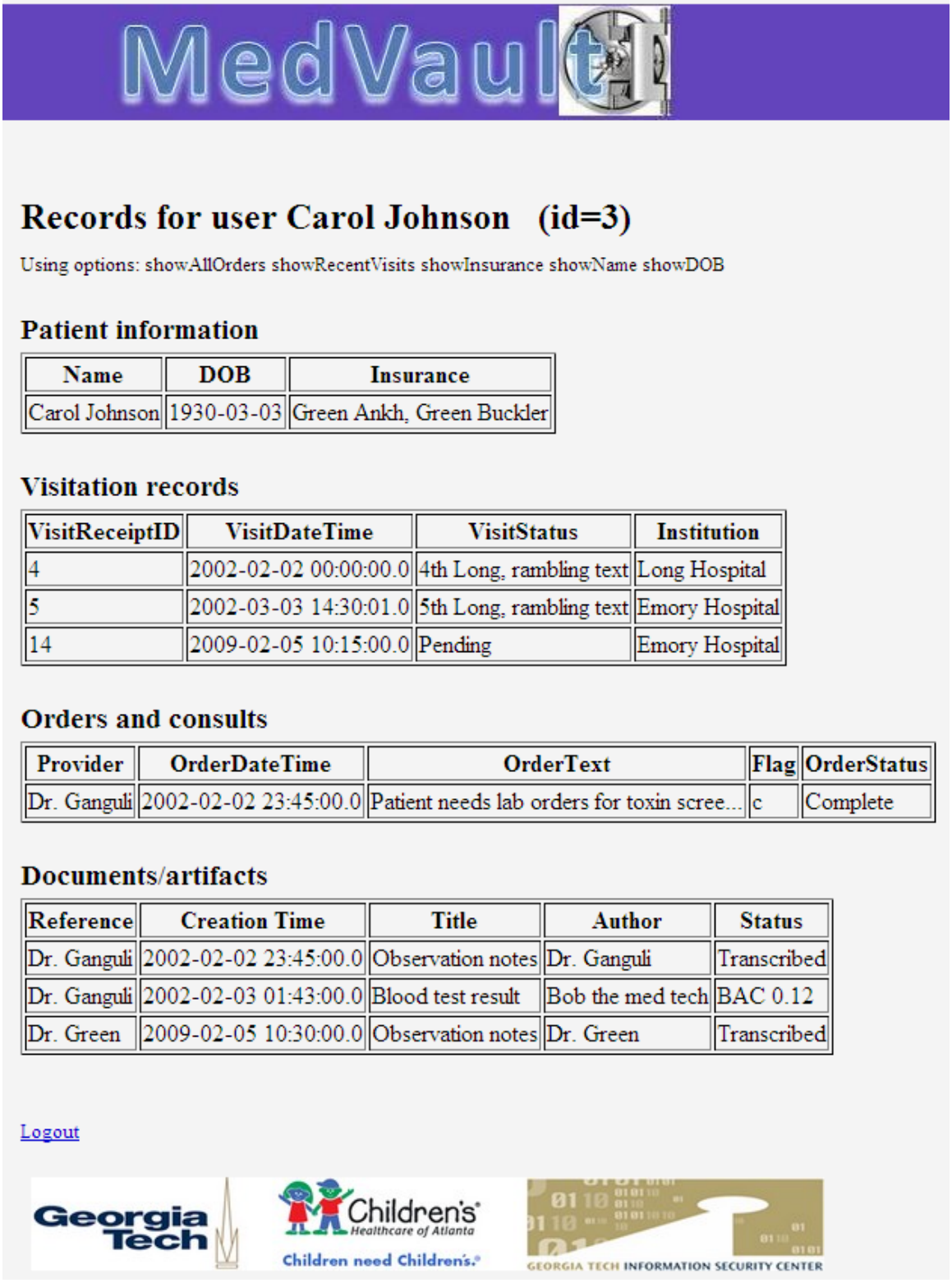


Figure 24: Screen shot of the doctor's view of the patient's record.

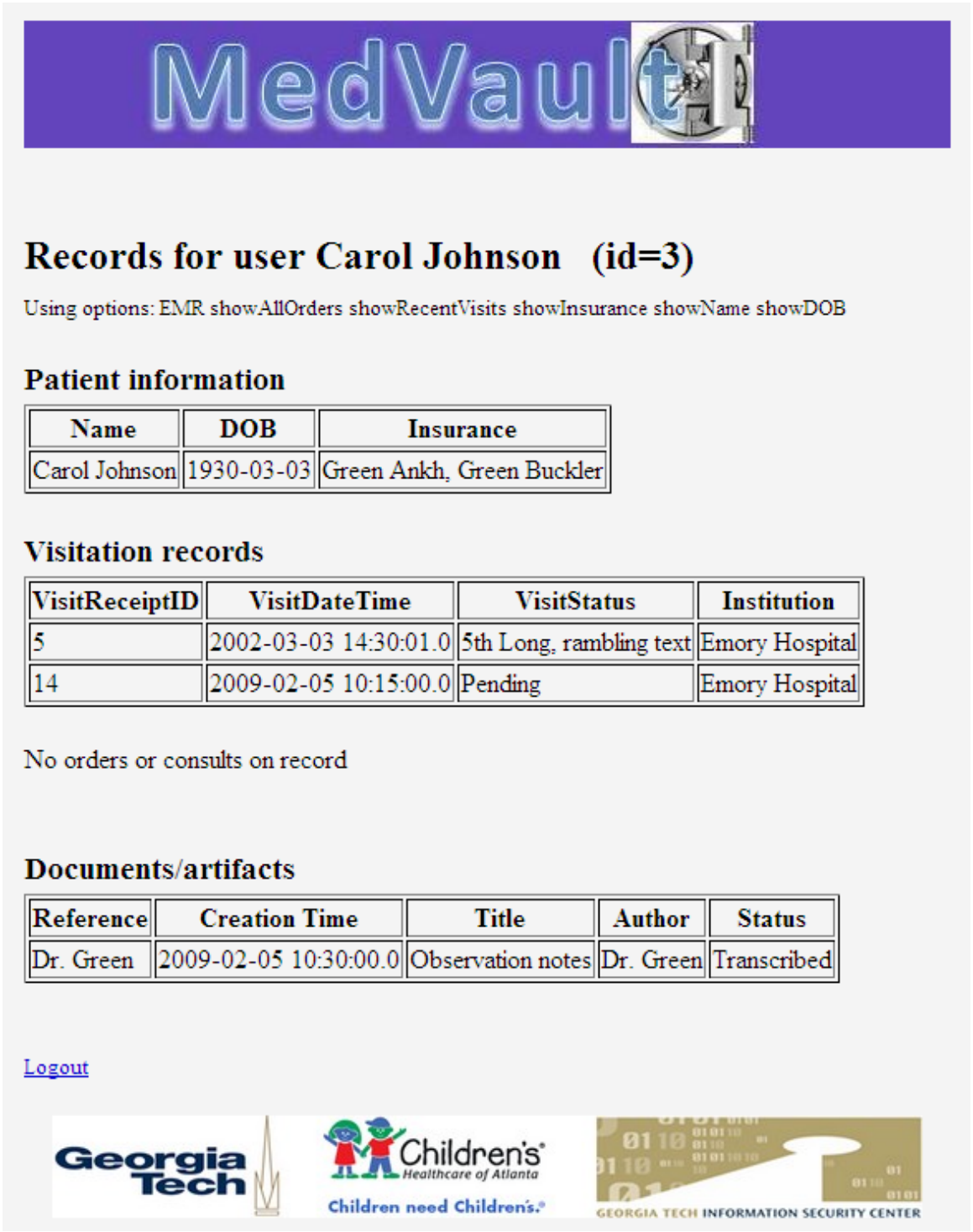


Figure 25: Screen shot of the EMT's view of the patient's record.

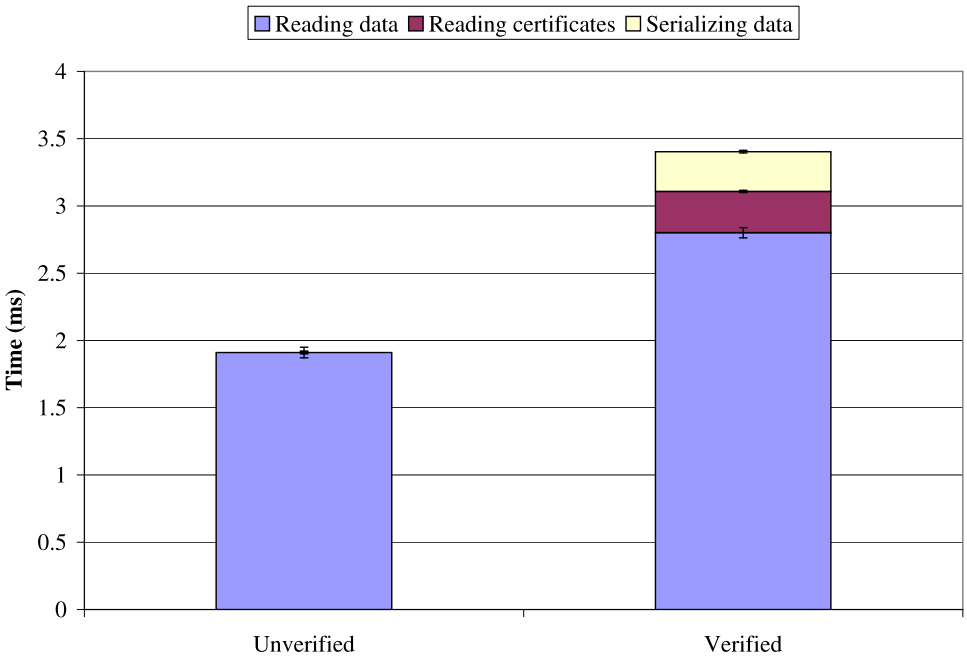


Figure 26: Comparison of reading source verified records and unverified records.

a single patient. (The pages were requested 150 times and only the last 7 timings were recorded, to give the JVM time to perform optimization.) The results are shown in Figure 26. For the source-verifiable case, the time to serialize the binary (Java object) message and the time to read in all needed certificates are shown separate from the base "reading data" time. For the plain case, the base "reading data" measurement includes the time to generate HTML from the records. The overhead ended up being a significant 78%, but the actual time overhead was trivial, at under 2ms.[9]

[9]The operation of reading and assembling an individual's records from the database is so fast that the execution time of the source-verified case was originally dominated by simply looking up the column headings. MySQL generates a temporary file on disk for this operation.

CHAPTER VII

CONCLUSION

> In the United States, the concept of "expectation of privacy" matters because it's the constitutional test, based on the Fourth Amendment, that governs when and how the government can invade your privacy.... The problem is, in today's information society, that definition test will rapidly leave us with no privacy at all. – Bruce Schneier [55]

In this work we are concerned about privacy with respect to all entities and not just the government. But the concept of the "expectation of privacy" still applies. If there are tools available to provide and protect privacy, then individuals will expect protection of their privacy. Due to the ease with which information technology can damage privacy through the mass accumulation, storage, and correlation of personal information, tools must be available to protect such information, in order to maintain the same expectation of privacy.

The research presented here works to protect individuals' privacy and security while preserving the benefits of an information society. Without such protection, users may reject advances in information technology, such as the use of sharable electronic health records.[14]

7.1 Contributions

Contributions to the art that have been described in this paper include

- A new construct for a credential that can efficiently store many claims and be combined with other credentials of the same construction. Combing credentials

can be done by a trusted "signing oracle", which does not need to even see the content of the credentials being combined.

- A measurement of the performance of the above credential, showing it to be at least two orders of magnitude faster than alternate credentials for reasonable parameters.

- A measurement of throughput on a server implementing the above credential for authentication, showing about 180 authentications per second (for showing 100 claims from a credential containing 2048), while an implementation of Brands' credential could only do about 2 authentications per second (for showing a credential with 20 claims).

- A novel problem area, concerning the selective disclosure of verified claims, when the original certifier has placed complex dependency requirements on the disclosure.

- A novel security model for the above problem area, where the owner of a credential and the relying party to which it is being shown conspire together to break usage constraints placed on the credential by its issuer.

- A solution to the above problem area, using graphs of hash-connected nodes.

- A measurement of the performance of the solution, showing it to be practical even for large sets of claims. The provided solution also has tight bounds on memory and computation, requiring no duplication or repeated processing of data items.

- A new system providing a user-agent based system for authentication and control of user credentials.

- Additional improvements have already been done to the above system by others, further increasing its security in the case of a physical device being compromised or stolen.

- A new framework for producing network-resident identity agents, with authentication, communications, and database functionality.

- A user-centric system for holding and securely sharing medical records, based on the above framework.

7.2 Limitations

7.2.1 Implementations

All implementations are prototype quality or should be considered prototype quality due to lack of external validation. The existing code is not suitable for production or widespread use, especially for security and/or privacy sensitive systems. Additionally, the APIs for all implementations are generally limited to functionality needed by existing code and are therefore incomplete.

Cryptography is used throughout the described projects. While the implementations should be neutral with respect to what Java cryptographic providers are available, in many places the Bouncy Castle cryptographic library is required. [34] The Bouncy Castle library is only actually needed for generating certificates. Unfortunately, due to incompatibilities between the Bouncy Castle cryptographic provider and the default Sun cryptographic provider, the forced use of a single provider was required. For technical reasons, the Bouncy Castle provider was chosen over the default provider.

7.2.2 Cycles in the dependency graph

As mentioned at the end of Section 4.2.1, the implementation of credentials on claims with dependencies deals with cycles in the input by combining together all nodes in

a cycle into a single big node. This *can* introduce false requirements, by taking out branches of ORs. For example, consider the simple case "claim 1 depends on either claim 2 or claim 3, and claim 2 depends on claim 1". Claim 1 and claim 2 are in a cycle, so they will be combined into a single "claim 1,claim2" node. However, that node will depend on claim 3, when it doesn't need to, because it is self-satisfying. This is a limitation of the *implementation* and not of the approach. Handling of cycles is not considered a priority, and so they are removed in the simplest way possible that doesn't violate the release constraints. (Making the constraints stricter than they were meant to be is unfortunate, but not a violation of security of the system.)

7.3 Future Work

7.3.1 Merkle Hash-Tree Credential

Further improvements to the Merkle hash-tree based credential should be possible, including changes in the underlying representation and algorithms, optimizing the layout of claims in the tree, and more flexibility in how claims are shown. The current representation and traversal algorithms implemented assume that memory is not a limiting factor in handling the hash trees. However, there may be cases in which very large trees are used or the device used is memory constrained, in which case the tree should be constructed and traversed piece-wise. Previous work on this problem has focused on the case of generating the authentication path for a single claim or contiguous group of claims, because that is the most common usage [30, 63]. By contrast, the MHT credential requires generating authentication paths for claims nominally spread randomly. A second, related improvement would be to optimize the layout of the claims in the tree, so as to reduce the work needed to produce and verify the most commonly expected combinations of claims. Care must be taken, however, to ensure that unshown claims cannot be guessed by their location in the tree.

Both the Merkle hash-tree credentials and later protocols based upon them (i.e.,

GUIDE-ME) were influenced by the expectation that they would be used on low-power, hand-held devices. A comparison of the Merkle hash-tree credentials to other credentials on low-power devices would be helpful. Such comparisons will be complicated by the many possible variations, such as specific algorithms used (e.g., what hash algorithm for the hash-tree; what group to use for other credentials[1]) and dedicated hardware available.

7.3.2 Credentials with Dependencies

As discussed in Section 7.2.2, the current implementation of credentials on claims with dependencies handles cycles in a very simplistic manner. The specific addition of false requirements described seems easy to recognize and eliminate [2], but a formal analysis remains to be done. A harder problem is the introduction of false requirements simply from the nodes being combined, and thus only releasable as one combined claim. It is easy to show simple cases where solutions to this problem seem obvious, but also easy to show large cases where the simple cases result in an exponential blowup in the number of nodes needed. More analysis is needed, even if it leads only to a heuristic compromise.

Additionally, profiling of the implementation showed that the single low-level operation that took the largest percentage of run time was finding whether a node was a child or parent node of another node. In particular, during the creation of a graph, every time a node is added as a child or parent node it must first be verified that it is not a duplicate. Several approaches to better performance were tested, including using different structures (List, Set, combined List and Set) and implementations.[3] Different approaches were better or worse at different high-level operations, but none

[1] In particular, elliptical curves are usually used on resource-limited devices.

[2] If an OR condition has only one child node, remove that OR. The child node may have other parents, or it may become a top node.

[3] All performance data in in Section 4.4 is from a single, default implementation using the built in ArrayList class for the child and parent lists.

stood out. An implementation that uses different data structures at different times, puts more restrictions on the algorithms to remove the checks for duplicate links, or uses some other data structure could have greatly improved performance.

7.3.3 GUIDE-ME

There is much more work that can be done and has been done with the GUIDE-ME system. For example, [36] splits the user's private key into three shares, such that any two shares can provide a valid signature. One share is held by the local agent, one share is held by the remote agent, and one share is held on a portable token. If any of the three shares is compromised, it can be revoked by creating a new set of shares and updating the other two shares. Additionally, the user may store some credentials locally (since a compromise of the local device alone won't compromise the credential as a whole), and use the key share on the token to bypass the need for the remote agent.

Other variants and extensions were considered, but not implemented. First, if the remote agent is less trusted, it can store only encrypted credentials. A decryption key is then given to the relying party by the user when the credentials are shown. For security reasons, it only makes sense to have different claims in the credential encrypted under different keys. However, this adds additional overhead and complexity for key management, even if those keys are derived from a common key. Also, storing the credentials encrypted may hinder the ability of the remote agent to perform intelligent searches and maintenance.

Second, multiple remote agents can be used, each storing shares of credentials. While this is an obvious tactic, the shares must be generated at the claim-level. With the expectation of many small claims, this will probably be inefficient in general.

7.3.4 Zero-Knowledge-Proof Signed Records

An obvious protection against the release of signed medical records (or other sensitive credentials) is to use zero-knowledge proofs of the signature, for example, using the method suggested in Section 3.4.2. However, this also limits the signatures to being verifiable only to parties that communicate directly with the authorized record holder. While limiting the signatures in this way is good for protecting against the unauthorized disclosure of records, it may also prevent their legitimate use. Using zero-knowledge-proof signatures between trusted/federated domains may solve this problem.

APPENDIX A

SECURITY PROOFS FOR CREDENTIALS ON CLAIMS WITH RELEASE DEPENDENCIES

A.1 Threat model

The primary threat our system is designed to resist is the prover and verifier collaborating to cheat the certifier, by violating the dependencies on showing claims. In this case, security is done on a "can prove" basis, where it doesn't matter how the certified data is proven. In particular, the verifier does not have to follow established protocols or intentions. (Put simply, we do not assume that the verifier is an honest player in the system.) We refer to the verifier as suspicious (of the prover), but rule-breaking.

A.2 Definitions

Definition A *node* is an unambiguously parseable set of hash values plus a single claim. Each hash value is the hash of another node in the structure or an AND node secret share.

Definition A node is a *leaf node* if it is contained in the specially defined list of leaf nodes. If a leaf node's hash value can be found in any node in the structure, the entire structure must be considered invalid by the verifier.

Definition A non-AND node is *accepted* by a verifier if it is in the list of leaf nodes, or if its hash value is found in an already accepted node. An AND node is *accepted* if its hash is created by combining shares from already accepted nodes. A claim is *accepted* if it is in an accepted node.

Definition The *directed acyclic graph (DAG)* consists of all nodes reachable when starting from the list of leaf nodes.

Definition A task is called *possible* if and only if it is computationally feasible.

A.3 Forgery

Forgery covers all cases where the verifier is convinced that a claim is certified by a particular entity, when it was not. Forgery covers several different problems, depending on what part of the system is attacked. For example, a forger can try to attack the hash function to create a bad final or intermediate value. A secure hash function will prevent this type of attack.

Overview: the structure of the system is a signed DAG. It is shown that a verifier will only accept data if it was in the DAG when the DAG was created/signed and that a creator will only sign a DAG when all data in it is known.

Theorem A.3.1 *The verifier will only accept as valid data items that were known to be in the structure by its creator.*

A.3.1 Assumptions

- A collision-resistant hash function is used.
- The set of leaf nodes is fixed via a digital signature. The use of a redactable signature, such that not all leaf nodes are known to the verifier has no impact on our proof.
- The structure and type of a node is known and unambiguously parseable.
- The certifier will not put in hash values of nodes for which the contents are unknown.
- The certifier will not put a malformed node into the DAG.

- The "random" values used for AND nodes are generated by the certifier (and are not accepted from another entity).
- The hash function may be modeled as a random oracle [3] (for AND nodes).

A.3.2 Without ANDs

Proof First, by the collision resistance assumption on the hash function, it is not possible to create two nodes with the same hash value but different contents. By definition, a leaf node will be accepted if and only if its hash is in the list of leaf nodes and a non-leaf node will be accepted if and only if its hash is given in an already accepted node. By induction, a node will be accepted if and only if it is either in the list of leaf nodes or connected to a node in the list of leaf nodes via a chain of accepted nodes, i.e., it is in the DAG.

A node's value and therefore hash depends on the hash values of the nodes to which it links and the claim it contains. Therefore, a node's value cannot be computed without the hash of the nodes to which it links already being known. By induction, a node's value cannot be computed without the hash of all nodes 'down' from it in the DAG being known. Since this applies to all nodes, it applies to the leaf nodes. Thus, the set of leaf nodes cannot be computed without knowing the hash values of all other nodes in the DAG. By assumption, the creator of the structure will not accept the hash value of a node without also knowing the contents. Thus, since the hash values of all nodes in the DAG must be known, the contents of all nodes in the DAG must be known. And since the verifier will only accept nodes in the DAG, the verifier will only accept nodes known to be in the structure at creation time, and thus data items that were in the structure at the time of its creation.

A.3.3 With ANDs

Lemma A.3.2 *Given a secure hash function H that may be modeled as a random oracle, a list of values $\{x\}$ in the domain of H such that $|\{x\}| <<< |H|$, it is not*

computationally feasible to find a set of values $\{y\}$ *such that* $x' = H(\{y\})$ *xor* $\{y\}$*, where* x' *is any element of* $\{x\}$*.*

Proof $H(\{y\})$ is a completely different random value for each different $\{y\}$. XORing in non-random (but chosen ahead of time) values can't make the output any less random. Thus, $H(\{y\})$ xor $\{y\}$ is still a random value for each different $\{y\}$. The probability of such a random value appearing in $\{x\}$ is $|\{x\}|/|H|$, which is very small in a practical system.

Proof By the assumptions of a secure hash function and a reasonable generation of (pseudo) random values for AND nodes, it is not possible to find a node that hashes to any of the "random" values. Similarly, it is not possible to find a node that hashes to the parity value for an AND node. Thus, it is not possible to fake a non-AND node from the values and constructions provided for AND nodes. An AND node itself has a specific form. By the assumption of unambiguous nodes, an AND node cannot be mistaken for a non-AND node. By the definition for accepting an AND node and the assumption that a certifier will not put a malformed node into the DAG, an AND node is accepted if and only if the self-check values contained in it and the parity value are all given in already accepted nodes. (Since it is trivial to forge an AND node if one of the check values may be freely determined, a sincere verifier cannot accept an unverified/unaccepted check/parity value.) The case of forging an AND node using only accepted values is equivalent to the case given in Lemma 1, but with $\{y\}$ constrained to come from the same list as $\{x\}$. As such, by Lemma 1 it is not possible to forge an AND node from already accepted values. Since it is not possible to forge either an AND or non-AND node using the AND node constructs, the addition of AND nodes does not allow any forgery not already possible. But, it was already shown that in the case with no AND nodes, it is not possible to get any forgery accepted.

A.4 Loss of Privacy

A loss of privacy occurs when the verifier learns something that the prover doesn't mean to show.

In the system, there is only one place where additional information is given out: the hash value of some nodes that the prover doesn't show will still be given to the verifier. This could allow an attacker to perform a dictionary attack against a node, except that random padding makes such an attack infeasible.

A.5 Violation of Dependencies

A violation of dependencies occurs when the prover can convince a suspicious, but rule-breaking verifier that the creator certified a claim (which is in the structure), without respecting the dependencies that the signer expected to be enforced.

A.5.1 Additional Assumptions

- The "random" values must look random, such that it is not possible to find a (simple) generator that will produce the AND values.

A.5.2 Without ANDs

Proof From the previous proof against forgery, it is already known that an attacker cannot get a verifier to accept a node which was not in the DAG when the creator signed it. A node (in the DAG) will not be accepted by a verifier unless the hash of that node is given in an already accepted node, a leaf node, or an accepted partial node. An accepted partial node is the case where a prover can convince a verifier that a bit of data is part of a valid node, without showing the whole node. Under the random oracle model of a hash function, it should not be possible to convince a verifier to accept any partial node, because

1. The only evidence a verifier should accept is a hash value

2. The hash value can only be computed by the random oracle given the entire node

3. Therefore, a verifier cannot check or accept a partial node

Using a real hash function that processes data such that later data cannot be processed without earlier data (or intermediate values from already processed earlier data), it is easy to see that even using intermediate values from the hash calculation can't allow a prover to make malicious use of partial nodes, because the data that the node is showing is the last part of the node to be hashed. Virtually all hash functions meet this criteria, with the exception being hash functions using tree-based hashing (such as MD6 [51] or the optional mode of Skein [22]). With partial nodes excluded, only hash values in already accepted nodes or leaf nodes will be accepted by a verifier as evidence of a node belonging in the DAG. Additionally, since such accepted nodes or leaf nodes cannot be accepted by the verifier without knowing their contents, the dependencies upon them are maintained.

A.5.3 With ANDs

Proof The new assumption is necessary to prevent a prover from convincing the verifier to accept a "random" value for use in an AND node that has not been shown in an already accepted or leaf node. If the values chosen form an obvious pattern, or are generated by a weak pseudo-random number generator, the verifier may accept a confirmed pattern as sufficient evidence to accept a value.

Under this random-looking assumption, the verifier will only accept "random" values that appear in already shown and accepted nodes or leaf nodes. By the assumption that the certifier generates the "random" values and the unforgability of AND nodes, those "random" values will be values intended as random values by the prover. It was shown above that in this situation, the verifier can only show nodes in the DAG. As the verifier cannot know that a node is in the DAG without the full

path going back to leaf nodes, the difficulty of violating the dependencies reduces to the difficulty of having a forged node accepted by the verifier or getting a partial node accepted. Both of these were already shown to not be possible.

APPENDIX B

SECURITY PROOF OUTLINE FOR GUIDE-ME PROTOCOLS

B.1 Assumptions

We assume that honest parties follow the protocol, as described in Section 5.4.1. Honest parties will reject messages with bad session IDs, messages with bad signatures, and (in the case of a relying party) messages that lists a wrong public key. Honest parties will generate their own nonces and will not re-use a nonce.

We assume that all cryptography used is secure and may be treated as a black box.

We assume that the local and remote agents are not compromised. (Dealing with a compromised agent is a separate topic.)

B.2 Replay attacks

GUIDE-ME uses nonces to define a session. Each party selects their own nonce; the two nonces concatenated together define a session. The chance of a single, correctly generated random nonce repeating is small enough to be ignored. [1] Replay attacks against GUIDE-ME can take two forms: attacks on a still-active session, and attacks repeating messages from a no longer active session.

For a still-active session, the most that replaying a message will achieve is to generate another copy of the response. [2]

[1] GUIDE-ME generates 16 byte random nonces using Java's SecureRandom class. If a million-trillion nonces were created, the chance of a repeat would still be under 50%.

[2] This is for the GUIDE-ME specific protocols. If GUIDE-ME is used as part of a larger protocol where repeated messages can cause problems, that is the responsibility of the larger protocol to handle.

For a non-active session, replaying a message can get a response from the remote agent—another copy of the response it sent the first time—but this response seems useless, since no relying party will accept it, and only the original replying party can decrypt it. Neither the user nor relying party will accept an out-of-session message.

B.3 Man-in-the-middle attacks

The resistance of GUIDE-ME to man-in-the-middle attacks (and weaker attacks) comes primarily from two components of the protocol. First, the user's public key is directly tied to the credential and claims being shown via a trusted signature from an identity provider. This cannot be matched by protocols such as SSH or TLS/SSL, without a predetermined relationship (such as a saved certificate or public key). Second, the user proves ownership of the matching private key by tying the showing of the credential to the current session (through the session ID) and the specific relying party (through the relying party's public key).

In order to provide a stronger security model, a man-in-the-middle attacker may be assumed to be able to successfully impersonate a low-security relying party. Doing so will net the attacker a valid partial credential (showing) and other valid messages. We consider the privacy loss through claims freely given to the attacker in this way to be small, and not a valid attack. (Even this loss can be prevented via two-way trust negotiation, but that is beyond the scope of this discussion.)

A man-in-the-middle attacker cannot impersonate the user or user's remote agent, even if the attacker has one or more separate, valid authentication sessions with the user. To impersonate the user's remote agent, an attacker requires the either the remote agent's private key or a fake message from the user listing a different public key for the remote agent. Capturing the remote agent's private key is equivalent to compromising it, which was assumed not to happen. Faking a message from the user is equivalent to compromising the local agent (or user's private key), which was

assumed not to happen. Impersonating the user without just replaying messages from the user also requires faking a message, and thus compromising the local agent (or user's private key). Alternately, the attacker could send an invalid message, such as being signed with a different key than the credential is issued to, having a bad signature, or not being signed at all. An honest relying party will reject all of these invalid messages.

A man-in-the-middle cannot get an honest relying party to accept altered messages, or messages meant for a different party. The user's message to the relying party includes both the session ID and the relying party's public key. The session ID limits the scope of the message to the active session, while the relying party's public key specifies exactly who the user thinks the relying party is. Messages to the relying party (from both the user and the remove agent) are signed using the relying party's public key, and cannot be altered with without detection. By the defined assumptions on an honest party, an honest relying party will not accept messages it detects as having been altered.

APPENDIX C

PUBLICATIONS

The following publications resulted from this research:

- **David Bauer**, Douglas M. Blough, Apurva Mohan, "Redactable Signatures on Data with Dependencies and their Application to Personal Health Records", to appear Proceedings of 2009 ACM workshop on Privacy in the electronic society, Nov 9, 2009, Chicago, IL, USA.

- **David Bauer**, Douglas M. Blough, David Cash, "Minimal information disclosure with efficiently verifiable credentials", Proceedings of the 4th ACM workshop on Digital identity management, October 31, 2008, Alexandria, VA, USA.

- Federica Paci, **David Bauer**, Elisa Bertino, Douglas M. Blough, Anna Squicciarini, "Minimal credential disclosure in trust negotiations", Proceedings of the 4th ACM workshop on Digital identity management, October 31, 2008, Alexandria, VA, USA.

- Apurva Mohan, **David Bauer**, Douglas M. Blough, Mustaque Ahamad, Bhuvan Bamba, Ramkumar Krishnan, Ling Liu, Daisuke Mashima, Balaji Palanisamy, "A Patient-centric, Attribute-based, Source-verifiable Framework for Health Record Sharing", GIT CERCS Technical Report No. GIT-CERCS-09-11, 2009, Georgia Institute of Technology, Atlanta, GA, USA.

- Federica Paci, **David Bauer**, Elisa Bertino,Douglas M. Blough, Anna Squicciarini, and Aditi Gupta, "Minimal Credential Disclosure in Trust Negotiations", Identity in the Information Society, to appear.

REFERENCES

[1] American Civil Liberties Union, "Anti-Real ID legislation in the states," 2009.

[2] Bayer, D., Haber, S., and Stornetta, W., "Improving the efficiency and reliability of digital time-stamping," in *Sequences II: Methods in Communication, Security, and Computer Science*, pp. 329–334, Springer-Verlag, 1993.

[3] Bellare, M. and Rogaway, P., "Random oracles are practical: a paradigm for designing efficient protocols," in *CCS '93: Proceedings of the 1st ACM conference on computer and communications security*, (New York, NY, USA), pp. 62–73, ACM, 1993.

[4] Bellare, M. and Rogaway, P., "Optimal asymmetric encryption," *Lecture Notes in Computer Science*, vol. 950, pp. 92–111, 1995.

[5] Benjamin Grosof, e. a., "The role of intelligent agents in the information infrastructure." IBM White Paper, 1995.

[6] Bhargava, R., "Deploy CardSpace on your site without a SSL certificate," September 2007.

[7] Brands, S., "A technical overview of digital credentials," 1999.

[8] Brands, S., *Rethinking Public Key Infrastructures and Digital Certificates; Building in Privacy*. MIT Press, 2000.

[9] Brands, S., "Credentica - U-Prove SDK," 2007.

[10] Brands, S., Demuynck, L., and Decker, B. D., "A practical system for globally revoking the unlinkable pseudonyms of unknown users," *Information Security and Privacy*, pp. 400–415, 2007.

[11] Camenisch, J. and Herreweghen, E. V., "Design and implementation of the idemix anonymous credential system," in *Proceedings of the 9th ACM conference on Computer and communications security*, ACM PRESS, 2002.

[12] Camenisch, J., Hohenberger, S., and Lysyanskaya, A., "Compact e-cash," in *Advances in Cryptology — EUROCRYPT '05* (Cramer, R., ed.), vol. 3494 of *Lecture Notes in Computer Science*, pp. 302–321, 2005.

[13] Camenisch, J. and Lysyanskaya, A., "An efficient system for non-transferable anonymous credentials with optional anonymity revocation," in *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques: Advances in Cryptology*, Springer-Verlag, 2001.

[14] Cameron, K., "The laws of identity." Microsoft Web Services Technical Articles, 2005.

[15] Cates, J., "Robust and efficient data management for a distributed hash table," Master's thesis, Massachusetts Institute of Technology, May 2003.

[16] Chappell, D., "Introducing Windows CardSpace." Windows Vista Technical Articles, April 2006.

[17] Chaum, D., "Security without identification: transaction systems to make big brother obsolete," *Communications of the ACM*, vol. 28, Oct 1985.

[18] Chaum, D. and Evertse, J.-H., "A secure and privacy-protecting protocol for transmitting personal information between organizations," in *Advances in Cryptology*, pp. 118–167, Springer-Verlag, 1987.

[19] Chaum, D., Evertse, J.-H., and van de Graaf, J., "An improved protocol for demonstrating possession of discrete logarithms and some generalizations," in *Advances in Cryptology, Proceedings of Eurocrypt 87*, 1987.

[20] Dierks, T. and Rescorla, E., "RFC 5246: The transport layer security (TLS) protocol version 1.2," August 2008.

[21] Doctorow, C., "Identity theft: Our dodgy love affair with utility bills will end in tears." guardian.co.uk, August 2008.

[22] Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., and Walker, J., "The Skein hash function family." Submission to NIST, 2008.

[23] Fitzpatrick, B., "OpenID and SixApart." http://brad.livejournal.com/2226738.html, May 2006.

[24] Franklin, S. and Graesser, A., "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents," in *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)*, vol. 1193, (Berlin, Germany), Springer-Verlag, 1996.

[25] Goldreich, O., Goldwasser, S., and Micali, S., "How to construct random functions," *J. ACM*, vol. 33, pp. 792–807, 1986.

[26] Goyal, V., Pandey, O., Sahai, A., and Waters, B., "Attribute-based encryption for fine-grained access control of encrypted data," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 89–98, ACM, 2006.

[27] Hardt, D., "Identity 2.0." OSCON 2005, 2005.

[28] Hu, J. and Weaver, A. C., "A dynamic, context-aware security infrastructure for distributed healthcare applications," in *Pervasive Security, Privacy and Trust (PSPT2004)*, August 2004.

[29] International Organization for Standardization and International Telecommunications Union, "Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks." ISO Standard 9594-8:2001, March 2000.

[30] Jakobsson, M., Leighton, T., Micali, S., and Szydlo, M., "Fractal Merkle tree representation and traversal." RSA Security Conference, 2003.

[31] Johnson, R., Molnar, D., Song, D. X., and Wagner, D., "Homomorphic signature schemes," in *Topics in Cryptology – CTRSA 2002*, vol. 2271, pp. 244–262, Springer-Verlag, 2002.

[32] Jonsson, J. and Kaliski, B., "Public-key cryptography standards (PKCS) #1: RSA cryptography specifications version 2.1," 2003.

[33] Junior Achievement and The Allstate Foundation, "Teens and personal finance 2008," 2008.

[34] Legion of the Bouncy Castle, "Bouncy castle crypto API." http://www.bouncycastle.org, 2009.

[35] Lewis, J., "H.R.1268, Division B: REAL ID Act of 2005," 2005.

[36] Mashima, D., Ahamad, M., and Kannan, S., "User-centric handling of identity agent compromise," in *European Symposium on Research in Computer Security (ESORICS 2009)*, 2009.

[37] Mayer, M., "Google Health, a first look." `http://googleblog.blogspot.com/2008/02/google-health-first-look.html`, February 2008.

[38] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A., *Handbook of Applied Cryptography*. CRC Press, 1996. Section 10.4.3.

[39] Merkle, R., "A certified digital signature," in *Advances in Cryptography*, pp. 218–238, Springer-Verlag, 1989.

[40] Microsoft, "Microsoft's vision for an identity metasystem," 2005.

[41] Microsoft Inc., "Microsoft HealthVault." http://www.healthvault.com/, 2009.

[42] National Institute of Standards and Technology, "FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2," tech. rep., Department of Commerce, August 2002.

[43] Novell Inc, "Novell Identity Manager," 2008.

[44] Office of the National Coordinator for Health Information Technology, "Emergency responder electronic health record detailed use case," December 2006.

[45] Office of the National Coordinator for Health Information Technology, "Consumer empowerment: Consumer access to clinical information detailed use case," June 2007.

[46] Oracle, "Oracle Identity Management," 2007.

[47] Random House, ed., *Random House Unabridged Dictionary*. Random House, Inc., 2006. credential.

[48] Random House, ed., *Random House Unabridged Dictionary*. Random House, Inc., 2009. information.

[49] Random House, ed., *Random House Unabridged Dictionary*. Random House, Inc., 2009. knowledge.

[50] Recordon, D. and Fitzpatrick, B., "OpenID authentication 1.1," May 2006.

[51] Rivest, R. L., "The MD6 hash function – a proposal to NIST for SHA-3." Submission to NIST, 2009.

[52] Sadeh, N., Gandon, F., and Kwon, O. B., "Ambient intelligence: The MyCampus experience," Tech. Rep. CMU-ISRI-05-123, Carnegie Mellon University, July 2005.

[53] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C., "Role-based access control models," *IEEE Computer*, vol. 29, pp. 38–47, August 1996.

[54] Schneier, B., "Strong laws, smart tech can stop abusive 'data reuse'," *Wired.com*, June 2007.

[55] Schneier, B., "It's time to drop the 'expectation of privacy' test," *Wired.com*, March 2009.

[56] Seamons, K. E., Winslett, M., and Yu, T., "Limiting the disclosure of access control policies during automated trust negotiation," in *In: Network and Distributed System Security Symp. (NDSS 2001). Internet*, Society Press, February 2001.

[57] Shibboleth Project, "Project." http://shibboleth.internet2.edu/project.html, 2009.

[58] Squicciarini, A., Bertino, E., Ferrari, E., Paci, F., and Thuraisingham, B., "PP-trust-X: A system for privacy preserving trust negotiations," *ACM Transactions on Information and System Security*, vol. 10, 07 2007.

[59] Sun Microsystems, "Sun is first to market with integrated identity auditing and user provisioning system." http://www.sun.com/smi/Press/sunflash/2006-09/sunflash.20060911.1.xml, September 2006.

[60] Tarjan, R., "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[61] TARR, "Digital credential." http://tarr.uspto.gov/servlet/tarr?regser=serial&entry=76440329, January 2008.

[62] U.S. Department of Veterans Affairs (VA), "Veterans health information systems and technology architecture (VISTA)," 1996.

[63] Vaudenay, S., "One-time identification with low memory," in *Eurocode 92*, no. 339 in CISM Courses and Lectures, (Wien), pp. 217–228, Springer-Verlag, Berlin Germany, 1992.

[64] Wagner, D., "A generalized birthday problem," in *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, pp. 288–303, Springer-Verlag, 2002.

[65] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A., "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, (Boston, MA), pp. 255–270, USENIX Association, Dec. 2002.

[66] Wikipedia, "Privacy." http://en.wikipedia.org/w/index.php?title=Privacy&oldid=305126872, July 2009.

[67] Winsborough, W. H. and Li, N., "Safety in automated trust negotiation," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 3, pp. 352–390, 2006.

[68] Winsborough, W. H., Seamons, K. E., and Jones, V. E., "Negotiating disclosure of sensitive credentials," in *2nd Conference on Security in Communication Networks*, 1999.

[69] Ylonen, T., "RFC 4252: The secure shell (SSH) authentication protocol," January 2006.